Università degli Studi di Torino

Dipartimento di Informatica

Tesi di Laurea Magistrale in Informatica

Generation of histopathological tissue with Generative Adversarial Network

Relatori:
Prof. Roberto Esposito
Prof. Marco Grangetto

Candidato:
Davide Rubinetti

Correlatori:
Dott. Carlo Alberto Maria Barbano
Dott. Enzo Tartaglione

Anno Accademico 2020/2021
Sessione Ottobre 2021

Acknowledgements

At first I would like to reserve a special thank to Carlo Alberto Barbano, he has been the person who supported me the most and without his availability it would not have been possible to complete this work. I would like to thank the members of the Department of Computer Science of the University of Turin who supervised me: Enzo Tartaglione, Roberto Esposito, Marco Grangetto and Attilio Fiandrotti. I would also like to thank the contribution of Luca Bertero and Paola Cassoni, from the Medical Sciences Department of the University of Turin.

Special thanks are reserved also to my friend and colleague Davide Di Luccio, it has been a pleasure to work with you during these years and our friendship will continue also outside the university.

I would like to thank my parents for their support and for giving me the opportunity to study at the University. And I would also like to thank Asia, the person with whom I have been sharing my life for more than 6 years now, and who supported me in times of difficulty.

Finally, I thank everyone who supported me during this journey: professors, friends, relatives, colleagues and any other person with whom I shared this experience of life.

DICHIARAZIONE DI ORIGINALITÀ: "Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata."

Abstract

Colorectal cancer (CRC) is a type of cancer that begins in the large intestine (colon), the final part of the digestive tract. It typically affects older individuals, though it can occur at any age. It usually begins as small, noncancerous (benign) clumps of cells called polyps that form on the inside of the colon. Over time some of these polyps can become colon cancers. There are various parameters to determine the malignant potential of polyps that pathologists analyze: the type of polyps, their size and the degree of dysplasia. In this scope, a proper screening can help to find malignant polyps at an early stage, preventing their transformation into cancer.

In this work we study UniToPatho, a dataset of annotated high-resolution colorectal images, comprising different histological samples of colorectal polyps, collected from patients undergoing cancer screening.

Deep Learning based systems can help doctors in the delicate task of detecting and diagnosing the different types of colorectal polyps and their associated risk. In fact, Deep Learning techniques are able to get an extraordinary accuracy in medical pattern recognition, however they require large sets of annotated training images. So, the main goal of this work is to do data augmentation on UniToPatho, so that with a larger dataset it is expected that Deep Learning algorithms are more likely to get a higher accuracy. In particular, this work dealt with doing data augmentation by producing new samples of histopathological tissue starting from the semantic segmentation masks, using a particular kind of Generative Adversarial Network (GAN): a Pix2pix GAN. The idea is that if we generate new samples starting from the segmentation masks, we can produce highly precise, detailed and manageable outputs.

Contents

1 Introduction **7**
- 1.1 Colorectal cancer . . . 7
- 1.2 Aim of this work . . . 8
- 1.3 Overview . . . 9
- 1.4 Related works . . . 10

2 Neural Networks **12**
- 2.1 Machine Learning . . . 12
- 2.2 History of neural networks . . . 13
  - 2.2.1 Artificial Neuron . . . 13
  - 2.2.2 Perceptron . . . 14
  - 2.2.3 Multilayer Perceptron . . . 17
  - 2.2.4 Gradient Descent and Backpropagation . . . 18
  - 2.2.5 Loss functions . . . 21
  - 2.2.6 Deep Learning . . . 24
- 2.3 State-of-art network architectures . . . 27
  - 2.3.1 CNN . . . 27
  - 2.3.2 ResNet . . . 30
  - 2.3.3 U-NET . . . 31
  - 2.3.4 Generative Adversarial Network (GAN) . . . 33

3 Datasets **40**
- 3.1 PanNuke . . . 40
- 3.2 UniToPatho . . . 43

4 Segmentation **46**
- 4.1 The semantic segmentation task . . . 46
- 4.2 Training . . . 48
  - 4.2.1 Training . . . 49
- 4.3 UniToPatho segmentation . . . 50

5 Generation **52**
- 5.1 Pix2Pix GAN . . . 53
  - 5.1.1 Objective . . . 53

5.1.2 Discriminator 54
5.1.3 Generator 55
5.2 Training 56
5.2.1 Training on PanNuke 56
5.2.2 Training on UniToPatho 59

6 Results **70**
6.1 Segmentation 70
6.2 Generation 73
6.2.1 Results on PanNuke 73
6.2.2 Results on UniToPatho 74

7 Conclusions and future work **79**

List of Figures

1.1 Data contained in PanNuke: on the top there are the images of colon tissue, below their segmentation masks. Each color represents a specific label for the cells. 9

2.1 Artificial Neuron. Source: [32] 14
2.2 The step function. The x axis represents the input of the step function. 15
2.3 Linearly (a) and not linearly (b) separable datasets. 16
2.4 An example of multilayer perceptron: this MLP has 3 inputs, 2 outputs and its hidden layer has 4 hidden units. Source: [3] . . . 17
2.5 The error surface: the higher we are, the greater the error. By correcting the weights several times, we can undertake a "path" to arrive at a local minimum in the error surface. Source: [19] . . 19
2.6 Problems in setting a wrong learning rate value. Source: [4] . . . 19
2.7 Softmax Example Visualization. Source: [5] 23
2.8 A neural network learns features: the first layer learns to recognize edges and shapes; the second one identifies features like nose and eyes; the third one identifies a human face. These are representations obtained with a weighted linear combination of the previous layers filters. Source: [24] 25
2.9 Leaky ReLU function 26
2.10 MNIST database. Source: [6] 27
2.11 Convolutional Neural Network. We can identify the first part in which the features are learned while in the second one the classification is done. Source: [7] 28
2.12 An input image. It has a value for each pixel for three dimensions. Source: [7] 29
2.13 An input image and the filter. Passing the kernel over the image and multiplying the matrices we obtain the convolved feature. Source: [7] 30
2.14 Example of residual connection. It is a shortcut from x to the output block. Source: [34] p. 2 31
2.15 U-Net Architecture. The name derives from its particular U-shaped architecture. Source: [54] p.2 32

2.16 The convolution operation scheme. In the U-Net, the convolution with weights w and then the ReLU function are applied to the input feature map. Source: [8] 33
2.17 MaxPooling function. It takes the bigger value from the input feature map covered by the kernel. With a 2x2 kernel and stride 2 the resulting feature map has factor 2 lower spatial resolution. Source: [8] 34
2.18 Discriminative and generative models of handwritten digits. Source: [9] 36
2.19 Generative Adversarial Network for handwritten digits. 37
2.20 Human Faces generated by a StyleGAN. You can view other generated examples at `https://thispersondoesnotexist.com` . . . 39

3.1 Images of PanNuke. We can see the segmentation and classification of cells of 5 different classes in the tissues. Source: [29] . . . 41
3.2 An image of PanNuke (left) with the segmentation mask (right). Cells of different types are classified with different colours. 43
3.3 UniToPatho classification: there are six different classes in which the tissue is classified. Source [20] 44

4.1 Different tasks on images: the classification says what there is in the image; the semantic segmentation divides all differents objects; object detection detect each object; instance segmentation identifies each instance of objects. Source: [10] 47
4.2 Road scene image segmentation. Source: [11] 47
4.3 An image of PanNuke (left) with the segmentation mask (right). Cells of different types are classified with different colours. 48
4.4 U-Net++ architecture. Compared to the original version, this includes a series of nested dense convolutional blocks. In the graphical abstract, black indicates the original U-Net, green and blue show dense convolution blocks on the skip pathways, and red indicates deep supervision. Red, green, and blue components distinguish UNet++ from U-Net. Source: [64] 50

5.1 Some of image-to-image translation examples. Source: [38] 52
5.2 Training a conditional GAN to map edges to photo. The generator tries to fool the discriminator learning how to produce nice images starting from the input edge map. The discriminator tries to classify between fake and real images. Both discriminator and generator observe the input edge map. Source: [38] 54
5.3 An image of PanNuke and its own segmentation mask. Each color corresponds to a type of cell. Blue indicates neoplastic cells, green indicates inflammatory cells and red connective cells. White is used for the background. 57
5.4 The trend of the discriminator loss during the training on PanNuke. 60
5.5 The trend of the generator loss during the training on PanNuke. 61

5.6 Some examples of images of UniToPatho used to train the GAN. 62
5.7 On the left an original image of UniToPatho with the obtained segmentation mask. On the right the same image normalized with Torchstain, with the obtained segmentation mask. As you can see, cell segmentation is much better thanks to the normalization. 66
5.8 GAN produced after some epochs images affected by the "purple fog" effect. 67
5.9 The trends of the losses of the discriminator and of the generator when the "purple fog" was produced. On the x-axis there are the epochs, on the y-axis there are the loss values. When the L1 loss goes down (more or less at the epoch 60) the GAN starts to produce the "purple fog" effect. 68
5.10 The trends of the losses in the best training. 69

6.1 Example 1 of segmentation on PanNuke. 70
6.2 Example 2 of segmentation on PanNuke. 71
6.3 Example 1 of segmentation on UniToPatho. 71
6.4 Example 2 of segmentation on UniToPatho. 72
6.5 Example 3 of segmentation on UniToPatho. 72
6.6 Some synthetic images produced by the GAN trained on PanNuke. 73
6.7 Some examples where you can compare the real image and the synthetic image produced by the GAN starting from the segmentation mask. On the left there is the segmentation mask and beside the real image and the fake image. Have you tried to guess if the fake images are in the center or on the right? We hope that your choice was difficult to take. In the center there are the real images and on the right the fake images. 73
6.8 Some results obtained with the GAN trained on PanNuke. We took some images from the test set of UniToPatho and we ran the GAN in inference mode. 74
6.9 Some synthetic images produced by the GAN trained on UniToPatho. 75
6.10 Example 1 of comparison between real and synthetic image. . . . 76
6.11 Example 2 of comparison between real and synthetic image. . . . 76
6.12 Example 3 of comparison between real and synthetic image. . . . 76
6.13 Example 4 of comparison between real and synthetic image. . . . 77
6.14 Example 5 of comparison between real and synthetic image. . . . 77
6.15 Example 6 of comparison between real and synthetic image. . . . 77
6.16 Example 7 of comparison between real and synthetic image. . . . 78
6.17 Example 8 of comparison between real and synthetic image. . . . 78

Chapter 1

Introduction

1.1 Colorectal cancer

Colorectal cancer (CRC) is a type of cancer that begins in the large intestine (colon) which is the final part of the digestive tract. Tumors within the colorectum vary in their molecular, biological, and clinical features, and in their association with risk factors. For example, physical inactivity is associated with increased risk of cancer in the colon, but not in the rectum.
Colon cancer typically affects older individuals, though it can occur at any age. It usually begins as small, noncancerous (benign) clumps of cells called polyps that form on the inside of the colon. Over time some of these polyps can become colon cancers.

Polyps are common, detected in about half (including serrated polyps) of average-risk individuals 50 years of age or older undergoing colonoscopy, with higher prevalence in older age groups and among men compared to women. However, fewer than 10% of polyps are estimated to progress to invasive cancer, a process that usually occurs slowly over 10 to 20 years and is more likely as polyps increase in size.

There are different types of polyps which can vary in size, number, shape and other characteristics. Generally four types of polyps exist [1]: hyperplastic, neoplastic, hamartomatous and inflammatory. The main difference is their attitude to become a cancer. Hyperplastic polyps usually don't have malignant potential [27], so their risk is low, apart from some rare cases. Neoplastic polyps (usually called adenomas), instead, are more likely to become malignant and, eventually, turn into cancer. The most common types of adenomas are the tubular, tubulovillous, villous and sessile serrated [27].

There are various parameters to determine the malignant potential of polyps that pathologists analyze: the type of polyps, their size and the degree of dysplasia. In pathology, displasya is any of various types of abnormal growth or development of cells [2]. It is very important to know how to differentiate the level of dysplasia because the higher it is, the higher the risk of polyps of being malignant.

In 2020, there were estimated 104,610 new cases of colon cancer. Although the majority of CRCs are in adults ages 50 and older, 17,930 (12%) were diagnosed in individuals younger than age 50, the equivalent of 49 new cases per day.

Just as I am writing this introduction, the news came out that the legendary footballer Pele will be operated on for a suspected colon cancer.

1.2 Aim of this work

The main target of this project is to perform automatic detection and diagnosis using histopathological images. Colorectal polyps characterization relies on the histological analysis of tissue samples to determine the polyps malignancy and dysplasia grade. This allows to tailor patients management and follow up with the ultimate aim of avoiding or promptly detecting an invasive carcinoma.
In particular, in this thesis we focused on how to improve the performance of the models that carry out this work. One way to do this is to have a large dataset on which to train the models by doing data augmentation. In fact, with a larger dataset, it is expected that deep learning algorithms are more likely to get a higher accuracy. The reference dataset of this project is UniToPatho [20], a dataset of annotated high-resolution colorectal images, comprising different histological samples of colorectal polyps, collected from patients undergoing cancer screening.

Traditional data augmentation techniques can cause problems and unpleasant artifacts when applied to a medical dataset such as UniToPatho. For this reason, it has been decided to do data augmentation with a generative model, i.e. the goal is to train a generative model which is able to produce new very realistic synthetic samples thought to be included in the dataset. We want to generate new synthetic samples starting from the semantic segmentation masks. Basically, given an image, its relative semantic segmentation mask is an image where each pixel is labelled with a corresponding class of what is being represented. In our work, we process images containing cells of the colon tissue and we produce the semantic segmentation masks, where ideally the pixels of each cell are labelled depending on the type of that cell (for example inflammatory, neoplastic, epithelial, etc.).

Why was it decided to produce new samples starting from the semantic seg-

mentation masks? When we use a generative model, for example a GAN [31], we can generate new samples starting from a random input, i.e. a random noise. However, in this setting, the generated output is very often difficult to handle. Instead, starting from a semantic segmentation mask, we can generate images more precisely, in which each cell is drawn exactly as indicated in the segmentation mask.

1.3 Overview

After a brief overview of introduction to the world of neural networks and some of the main architectures, the datasets used will be analyzed in more details in the chapter 3.

Since UniToPatho does not contain for each image the labelled semantic segmentation mask, we trained a neural network able to produce these masks. We explain in chapter 4 how a neural network with the U-Net [54] architecture was trained on a dataset similar to UniToPatho: PanNuke, a dataset for the segmentation and classification of cells in the computational pathology field. Figure 1.1 shows some examples of images taken from PanNuke with the segmentation masks. Subsequently, the U-Net trained on PanNuke was used to produce the segmentation masks of the images of UniToPatho.

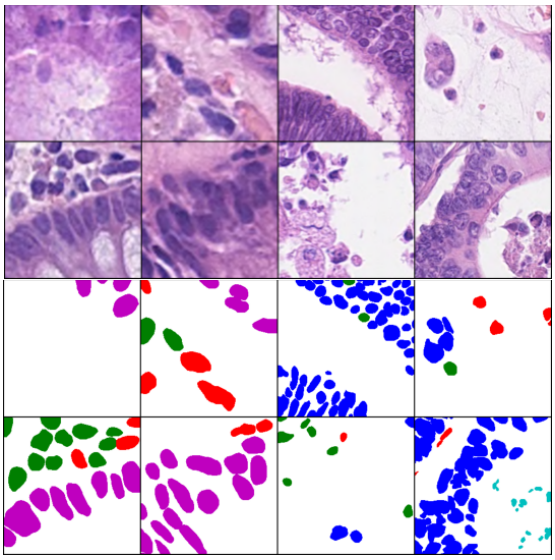


Figure 1.1: Data contained in PanNuke: on the top there are the images of colon tissue, below their segmentation masks. Each color represents a specific label for the cells.

Then, in chapter 5 we describe how we dealt with producing new samples starting from the segmentation masks using a Pix2pix Gan [38].

Finally, in chapter 6, we show the results that have been obtained in the segmentation and generation tasks.

1.4 Related works

In the scientific literature there are many works on colon cancer diagnosis that use various approaches. In fact, nowadays, digital technologies are used extensively to classify medical images using traditional methods of Machine Learning (ML) or those of Deep Learning (DL). In [59] Tsai et al. studied and tested several deep learning models for the recognition of colorectal cancer tissue using CNN. In the article they propose a series of parameters to improve the accuracy of the classification and, to verify its functionality, they use CRC histological images as an experimental dataset, comparing 5 of the most used DL models. The experiment done with a ResNet50 in this study achieved 99.69% accuracy in the NCT-HE-100k dataset of 100,000 histological images.

Kather et al. [41] used a series of textual descriptors to analyze a multi-class problem of tumor epithelium and simple stroma with 5000 histological images. In this article they used four classification methods: (1) the k-nearest neighbors algorithm (k-NN), (2) an SVM decision function for classifying all categories, (3) assemble decision tree models using the RUSBoost method (an approach for alleviating class imbalance), and (4) use a 10-fold cross validation to train the classifiers, without an explicit stratification approach. The results indicated that SVM was the best classification method, which achieved 87.4% accuracy over eight classes. Recently, the classification of tumor types has been found to be more accurate using the CNN classification method.

Ben Hamida et al. [22] compared some of state-of-art DL models for patch and pixel level classification of a sparsely annotated colorectal histopathological dataset. They faced also the problem of the limited number of existing methods that treat the high resolution and size of Whole Slide Images (WSIs). This reaserch shows that the patch-level colon cancer histopathological image classification combined with a fine-tuning stategy is a right way for the tumour classification tasks. But this is a method that performs well if patches are preselected to represent one class each. In fact, if the WSI includes small regions of neighbouring classes at each patch, the model fails the prediction. Also Zhou et al. [36] said that classical deep learning methods which analyze thousands of patches extracted from WSIs may cause loss of integrated information of image. So they developed a new method, which used only global labels to achieve WSI classification and localization of carcinoma by combining features from different magnifications of WSIs. The model was trained and tested using 1346 colorectal

cancer WSIs from the Cancer Genome Atlas (TCGA). Their method classified colorectal cancer with an accuracy of 94.6 %, which slightly outperforms most of the existing methods.

Rathore et al. in [53] have developed an end-to-end pipeline that includes gland segmentation, cancer detection, and then further breaking down the malignant samples into different cancer grades. First a gland segmentation method uses the organizational appearance of colon glands to set up boundaries and to segment internal glandular structures. Then a multi-scale feature extraction with gland-based features, local-patch-based features and image-based features extracts information from slides. Finally using an ensemble of various classifiers the prediction is done by probabilistic estimates. Results show that the ensemble classification methodology produces robust classification and reinforces the performance of individual classifiers by a significant margin.

In histopathological analysis, stain is used to highlight the different biological parts but, as there are many types, this can be a problem for learning models. Ben et al. in [21] propose to reduce performance variability by using consistent generative adversarial (CycleGAN) networks to remove staining variation, so they improve upon the regular CycleGAN by incorporating residual learning. First they train a model to perform segmentation on tissue slides from a single dataset, applying data augmentations to increase robustness to unseen data. Second, they evaluate and compare the segmentation performance on data from other datasets, both with and without applying the CycleGAN stain transformation. Their transformation method improves the overall Dice coefficient by 9% over the non-normalized target data and by 4% over traditional stain transformation.

In short, the experimental results demonstrate that artificial intelligence has a broad application in classifying colorectal cancer histology images, and models can be a great help for doctors, because they enable them to make suitable decisions in the diagnostic process.

Chapter 2

Neural Networks

In this chapter we will provide an overview about the history of neural networks. Works about neural networks date back to the 40s and 50s. The first artificial neuron was the *Threshold Logic Unit (TLU)*, or *Linear Threshold Unit* [61], first proposed by Warren McCulloch and Walter Pitts in 1943. The model was specifically targeted as a computational model of the "nerve net" in the brain. One important and pioneering artificial neural network that used the linear threshold function was the *perceptron* [55], developed by Frank Rosenblatt. This model already considered more flexible weight values in the neurons, and was used in machines with adaptive capabilities.
In the late 1980s, when research on neural networks regained strength, neurons with more continuous shapes started to be considered. The possibility of differentiating the activation function allowed the direct use of the gradient descent and other optimization algorithms for the adjustment of the weights.
Lastly, we will see briefly how Deep Learning is a modern variation which is concerned with a very high number of layers of bounded size, which permits practical applications and optimized implementations.

2.1 Machine Learning

When we talk about neural networks and Deep Learning, it is always good to remember that we are in the field of Machine Learning. Machine Learning is the systematic study of algorithms and systems that improve their knowledge or performance with experience.
Main aspects of Machine Learning are: features, tasks and models. The aim of Machine Learning is to study algorithms that, starting from the features, build suitable models that solve specific tasks. Nowadays, Machine Learning

has become ubiquitous: it is used to build models for solving lots of everyday life tasks of different nature. We can summarize Machine Learning as using the right features to build the right models that achieve the right tasks.
Machine Learning approaches are traditionally divided into:

- Supervised learning: The computer is presented with example inputs and their desired outputs, given by a "teacher", and the goal is to learn a general rule that maps inputs to outputs. Types of supervised learning algorithms include classification and regression.
- Unsupervised learning: No labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning algorithms take a set of data that contains only inputs, and find structure in the data, like grouping or clustering of data points.

2.2 History of neural networks

Although nowadays neural networks have quite detached from the human cognitive model, in its most general and original form, a neural network is a machine designed to model the way in which the brain performs a particular task. To achieve good performances, neural networks employ a massive interconnection of simple computing cells referred to as "neurons".

2.2.1 Artificial Neuron

The figure 2.1 shows the model of a neuron, which forms the basis for designing neural networks.
An artificial single neuron is an information-processing unit that is fundamental to the operations of a neural network. For a given neuron k, let there be $m+1$ inputs with signals x_0 through x_m and weights w_{k0} through w_{km}. Usually, the x_0 input is assigned the value $+1$, which makes it a *bias* input with $w_{k0} = b_k$. This leaves only m acutal inputs to the neuron: from x_1 to x_m. The output of the kth neuron is:

$$y_k = \varphi(\sum_{j=0}^{m} w_{kj} x_j) \tag{2.1}$$

where $\varphi(.)$ is the activation function and y_k is the output of the neuron.

The activation function of a neuron is chosen to have a number of properties which either enhance or simplify the network containing the neuron. Here we identify some basic activation functions:

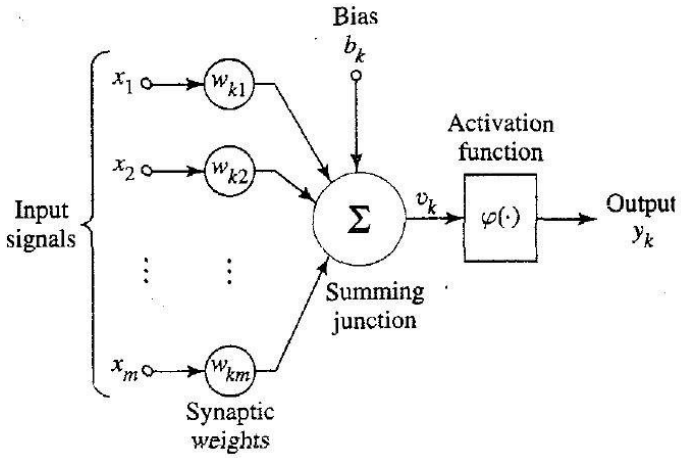


Figure 2.1: Artificial Neuron. Source: [32]

- Threshold function: defined as

$$\varphi(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{2.2}$$

- Sigmoid function: it ranges from 0 to 1 and its graph is s-shaped:

$$\varphi(x) = \frac{1}{1 + exp(-ax)} \tag{2.3}$$

  where a is the slope parameter of the sigmoid function.

- Hyperbolic tangent: it is similar in shape with the sigmoid function but ranges from -1 to 1:

$$\varphi(x) = \tanh x \tag{2.4}$$

- Rectifier Linear Unit (ReLU): for which it has been demonstrated to enable better training of deeper networks:

$$\varphi(x) = max(0, x) \tag{2.5}$$

2.2.2 Perceptron

The perceptron [55] is an algorithm for supervised learning of binary classifiers invented in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblatt. The perceptron is a linear classification algorithm. This means that it learns a decision boundary that separates two class using a line (called a hyperplane)

in the feature space. As such, it is appropriate for those problems where the classes are linearly separable. In the context of neural networks, a perceptron is an artificial neuron using the threshold function as the activation function.

The perceptron can be briefly described by a function that maps its input $\mathbf{x}$ (a real-valued vector) to an output value $f(\mathbf{x})$:

$$f(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.6}$$

where $\mathbf{w}$ is a vector of real-valued weights, $\mathbf{w} \cdot \mathbf{x}$ is the dot product $\sum_{i=1}^{m} w_i x_i$, m is the number of inputs to the perceptron and b is the bias. The idea behind this model is that the weight vector $\mathbf{w}$ can be progressively adjusted in order to produce a correct output.
The inputs are sent through a weighted sum function, then the weighted sum is sent through the threshold function and the output of the threshold function is the output of the perceptron. The output indicates the confidence of the prediction. The larger the numerical value of the output, the greater the confidence of the prediction.

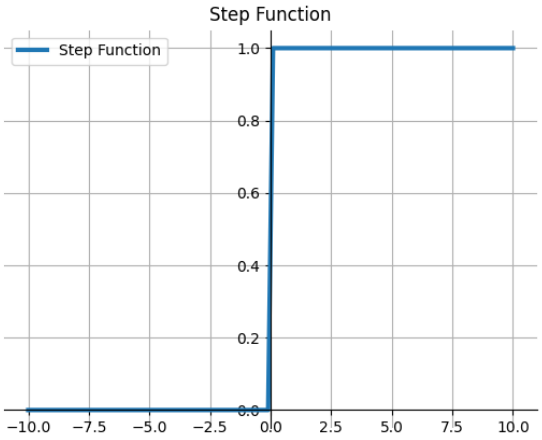
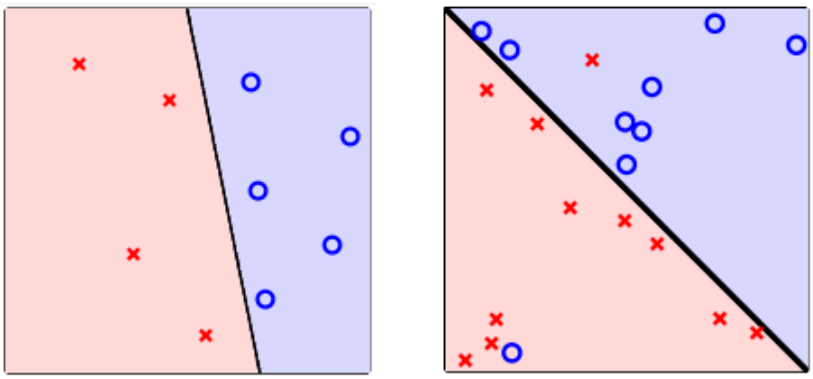


Figure 2.2: The step function. The x axis represents the input of the step function.

How do humans learn? We make a mistake, correct ourselves and, if lucky, make less mistakes. The perceptron algorithm computes the difference between the predicted value and the actual value. The difference is defined as an error. We can define the error E_j for the jth example $\boldsymbol{x_j}$ with the corresponding label y_j (either 0 or 1) as:

$$E_j = y_j - f(\boldsymbol{x_j}) \tag{2.7}$$



(a) (b)

Figure 2.3: Linearly (a) and not linearly (b) separable datasets.

Examples from the training dataset are shown to the model one at a time, the model makes a prediction, and error is calculated. The weights of the model are then updated to reduce the errors for the example. This process is repeated for all examples in the training dataset, called an epoch. This process of updating the model using examples is then repeated for many epochs.
The updating rule is defined as:

$$\mathbf{w} = \mathbf{w} + \eta(y_j - f(\boldsymbol{x_j}))\boldsymbol{x_j} \tag{2.8}$$

where η is the *learning rate*: it is a positive hyperparameter which determines the significance of each adjustement. Usually it is set between 0 and 1. If the learning rate is high, small errors can cause considerable shifts in the values of weights. On the contrary, if the learning rate is small, significant errors cause minimal changes in the weights. Therefore, it is necessary to find the right balance between the two extremes.

Training is stopped when the error made by the model falls below a certain level or no longer improves, or a maximum number of epochs is performed. If the training set is linearly separable, then the perceptron is guaranteed to converge. Furthermore, there is an upper bound on the number of times the perceptron will adjust its weights during the training [50].

In order to solve the problem of inseparable linearity using neural networks, we need to introduce a multilayer neural network, which we will describe in the next section.

2.2.3 Multilayer Perceptron

A multilayer perceptron (MLP) is a class of feedforward[1] artificial neural network. A MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. In a MLP, such as the one shown in figure 2.4, each neuron in a layer is connected to every neuron in the previous one. It can distinguish data that is not linearly separable. The basic idea with which a MLP can solve non-linear problems is that the hidden layer projects the input space in a new space where the problem can be solved.

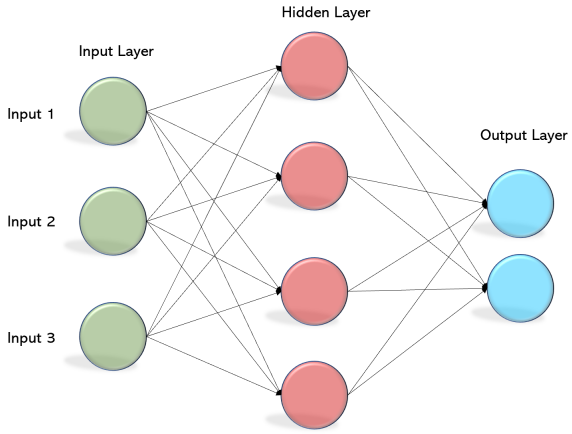


Figure 2.4: An example of multilayer perceptron: this MLP has 3 inputs, 2 outputs and its hidden layer has 4 hidden units. Source: [3]

MLPs can capture complex interactions among our inputs via their hidden neurons, which depend on the values of each of the inputs. We can easily design hidden nodes to perform arbitrary computation, for instance, basic logic operations on a pair of inputs. Moreover, for certain choices of the activation function, it is widely known that MLPs are universal approximators. Even with a single-hidden-layer network, given enough nodes (possibly absurdly many), and the right set of weights, we can model any function, though actually learning that function is the hard part. Moreover, just because a single-hidden-layer network can learn any function does not mean that you should try to solve all of your problems with single-hidden-layer networks. In fact, we can approximate many functions much more compactly by using deeper networks.

Similary to the perceptron, it is possible to define an error measure which can provide hints on how to adjust the weights of the network in order to obtain a

[1] A feedforward neural network is an artificial neural network wherein connections between the nodes do not form a cycle. As such, it is different from recurrent neural networks, where connections between neurons can form a cycle.

correct result. However, this is more complicated than the perceptron.

2.2.4 Gradient Descent and Backpropagation

In the *forward* pass, the neural network processes the input and computes its output. Then, the network tries to perfect its prediction by tweaking these weights. It does so, by comparing the predicted value $\hat{y}$ with the actual value of the example y in our training set and using a function of their differences. This function is called a loss function. For example, one of the most known error which has some nice algebraic properties is the *squared error*:

$$SE = \frac{1}{2}(y - \hat{y})^2 \tag{2.9}$$

We need to optimize weights to minimize the error, so, obviously, we need to check how the error varies with the weights. The idea of gradient descent is that the loss function creates an "error surface" in which we have to find a minimum value related to the weights of the network. To do this we need to compute the gradient of the error with respect to each parameter in the network. Considering a generic neuron j which has n weights, the gradient of the error is:

$$\nabla E(w_j) = \left[\frac{\partial E}{\partial w_{j1}}, \ldots, \frac{\partial E}{\partial w_{jn}} \right] \tag{2.10}$$

where E is the error and w_{ji} is the ith weight of the jth neuron. The key is that the *negative* of the gradient shows the directions along which the weights should be moved in order to optimize the loss function. So, this way the gradient guides the model whether to increase or decrease weights in order to minimize the loss function. For more than two parameters, visualization of the "error surface" is very hard and tricky. Figure 2.5 can give an idea of how it is made.

Finally, in the *backward* pass, the network error is used to update each weight by taking a step in the opposite direction of the gradient of E:

$$w_{ji} = w_{ji} + \Delta w_{ji} \tag{2.11}$$

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} \tag{2.12}$$

If we properly compute the gradients, we will note that the gradients of the neurons of a layer depend on the output of successive layers, so the corrections of the weights are made starting from the output level and, progressively, the weights of the previous layers are corrected, up to the input one. Details on the computation of gradients in a MLP are omitted for reasons of brevity, but can be found in [32].

How big the steps gradient descent takes into the direction of the local minimum are determined by the learning rate η, which figures out how fast or slow we

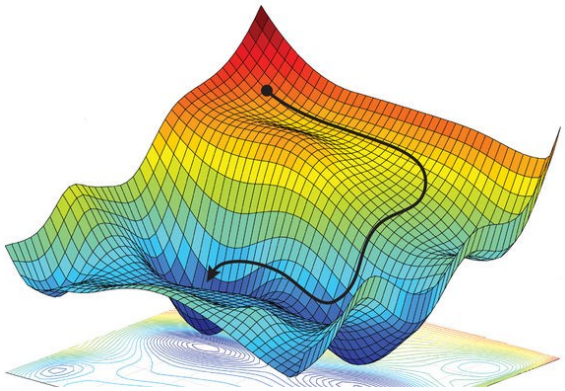


Figure 2.5: The error surface: the higher we are, the greater the error. By correcting the weights several times, we can undertake a "path" to arrive at a local minimum in the error surface. Source: [19]

will move towards the optimal weights.
To reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high. This is important because if the steps the network takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent. If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while. Figure 2.6 shows graphically the problems in setting a wrong learning rate value.

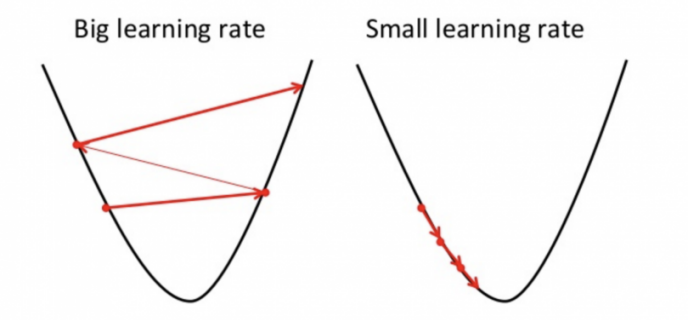


Figure 2.6: Problems in setting a wrong learning rate value. Source: [4]

For tackling this problem, many optimization techniques have been proposed. The most common optimization is the introduction of the momentum term α:

$$w_{t+1} = w_t + \Delta w_{t+1} \tag{2.13}$$

$$\Delta w_{t+1} = \alpha \Delta w_t - \eta \nabla E(w_t) \tag{2.14}$$

where α is a hyperparameter which accelerates the descent in steady downhill directions and has a stabilizing effect for directions that oscillate in time. An empirical value for α is usually set at $\alpha = 0.9$.

In addition to this, when training deep neural networks, it is often useful to reduce learning rate as the training progresses. This can be done by using pre-defined learning rate schedules or adaptive learning rate methods. Learning rate schedules seek to adjust the learning rate during training by reducing the learning rate according to a pre-defined schedule. Common learning rate schedules include time-based decay, step decay and exponential decay. The challenge of using learning rate schedules is that their hyperparameters have to be defined in advance and they depend heavily on the type of model and problem. Adaptive gradient descent algorithms such as *Adagrad* [46], *Adadelta* [63], *RMSprop* [35] and *Adam* [42] provide an alternative to classical SGD. These per-parameter learning rate methods provide heuristic approach without requiring expensive work in tuning hyperparameters for the learning rate schedule manually.

Types of Gradient Descent

There are three popular types of gradient descent that mainly differ in the amount of data they use:

- **Batch Gradient Descent**: it calculates the error for each example within the training dataset, but the model gets updated only after all training examples have been evaluated:

$$w_{ji} = w_{ji} + \sum_{x \in T} \Delta w_{ji}^{x} \tag{2.15}$$

  where T is the training set and Δw_{ji}^{x} is the contribution of example x for the correction of the weight w_{ji}.

- **Stochastic Gradient Descent (SGD)**: by contrast, stochastic gradient descent (SGD) applies the correction of the weights for each training example within the dataset, meaning it updates the parameters for each training example one by one.

- **Mini-Batch Gradient Descent**: the two previous methods present some disadvantages such that they are rarely used when training deep neural networks. Batch gradient descent takes too much time to converge, as it uses the entire dataset just to perform a single update step. Instead, SGD converges faster because it performs updates for each sample at a time but, on the other hand, this ultimately complicates convergence to the exact minimum, because once SGD reaches close to a local minima then it does not settle down, but bounces around.

Mini-batch gradient descent is a combination of the concepts of SGD and batch gradient descent. It simply splits the training dataset into small batches and performs an update for each of those batches. This creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. This is the method truly used for training deep neural networks. Usually the term SGD is used to refer to this method of training: in this work, when we talk about SGD, we will be referring to its mini-batch variant.

2.2.5 Loss functions

Previously, we said that a neural network produces the output, computes the relative error and then corrects its weights. But how does the network calculate the error? The method to calculate the error is called *loss function*. Most of the time, SGD is used in its mini-batch variant and the loss is calculated as the average error of n examples belonging to a mini-batch. Then gradients are computed and used to update the weights of the neural network. This is how a neural network is trained.

Lots of loss function have been studied in machine learning: there are many loss functions to choose from and it can be challenging to know what to choose, or even what a loss function is and the role it plays when training a neural network. There are various factors involved in choosing a loss function for specific problems such as the type of machine learning algorithm chosen, ease of calculating the derivatives and the percentage of outliers[2] in the dataset. Very often, the choice of the loss function strictly depends also on the nature of the task thought to be solved. For example, for solving a regression task, *MSE* or *L1 Loss* are usually used, while for classification tasks *Cross Entropy* is a valid alternative. Here, the most known and essential loss functions will be briefly described. In the following, we will use y_i and $\hat{y}_i$ to refer to the true value and to the prediction value of the ith example.

Mean Squared Error / Quadratic Loss / L2 Loss

As the name suggests, mean squared error is measured as the average of squared difference between predictions and actual observations. It is only concerned with the average magnitude of error irrespective of their direction. However, due to squaring, predictions which are far away from actual values are penalized heavily in comparison to less deviated predictions. Additionally, MSE has nice mathematical properties which makes it easier to calculate gradients. The formulation

[2] An outlier is an example that lies an abnormal distance from other examples in a dataset.

is:

$$L_{MSE} = \frac{\sum_{i=1}^{n}(y_i - \hat{y_i})^2}{n} \tag{2.16}$$

Mean Absolute Error / L1 Loss

Mean absolute error, on the other hand, is measured as the average of sum of absolute differences between predictions and actual observations. Like MSE, this as well measures the magnitude of error without considering their direction. Unlike MSE, MAE needs more complicated tools such as linear programming to compute the gradients. Plus MAE is more robust to outliers since it does not make use of square. The mathematical formulation is:

$$L_{MAE} = \frac{\sum_{i=1}^{n}|y_i - \hat{y_i}|}{n} \tag{2.17}$$

Cross Entropy Loss / Negative Log Likelihood

This is the most common setting for classification problems. Cross-entropy loss increases as the predicted probability diverges from the actual label.

Suppose we are facing a multiclass classification problem, so we have to predict the labels of the examples among k possible labels. For example, if an instance belongs to the third class, the target value will be a one-hot vector of length k as: $[0, 0, 1, 0, ..., 0]$. In order to face this problem, we might use in the output layer k neurons and just apply the sigmoid to all the output nodes so that we get values between 0–1 for all the outputs, but there is an issue with this. When we are considering probabilities for multiple classes, we need to ensure that the sum of all the individual probabilities is equal to one, since that is how probability distribution is defined. Applying sigmoid does not ensure that the sum is always equal to one, hence we need to use another activation function. The activation function we use in this case is *softmax*. This function ensures that all the output nodes have values between 0–1 and the sum of all output node values equals to 1 always. The formula for softmax is as follows:

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \tag{2.18}$$

where z_i is the input of the ith output neuron. Figure 2.7 shows a graphical example of the softmax application. Essentially, as you can see, softmax is just an exponential function which makes sure that outputs are all in the range of 0–1 and makes sure that the sum of all the output values equals to 1 (dividing each exponential with the sum of all exponentials).

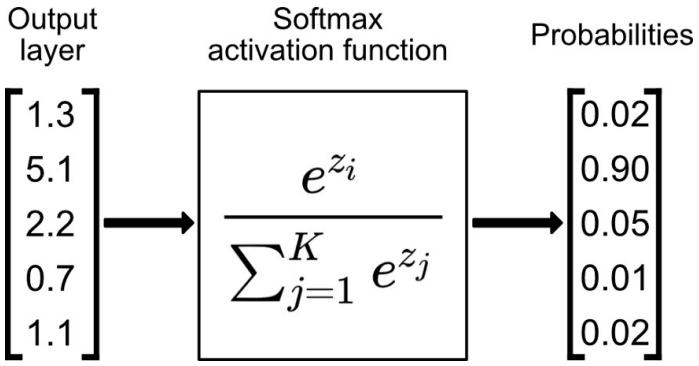


Figure 2.7: Softmax Example Visualization. Source: [5]

Now that our outputs are in a proper format, let's go ahead to look at how the cross-entropy loss is formulated:

$$L_{CrossEntropy} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) \tag{2.19}$$

In information theory[3], the cross-entropy between two probability distributions p and q over the same underlying set of events measures the average number of bits needed to identify an event drawn from the set if a coding scheme used for the set is optimized for an estimated probability distribution q, rather than the true distribution p. In these terms, cross-entropy of the distribution q relative to a distribution p is defined as follows:

$$H(p, q) = -\mathbb{E}_p[\log(q)] = -\sum_{x \sim p} p(x) \log(q(x)) \tag{2.20}$$

where $\mathbb{E}_p[.]$ is the expected value operator with respect to the distribution p and $x \sim p$ denotes a random variable x which has distribution p.

In the case of binary classification, labels are only 2 and *binary cross-entropy loss* is used:

$$L_{BinaryCE} = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \tag{2.21}$$

Note that when actual label is 1 ($y_i = 1$), second half of function disappears whereas in case actual label is 0 ($y_i = 0$) first half is dropped off. An important aspect is that cross entropy loss penalizes heavily the predictions that are *confident but wrong.*

[3]Information theory is a branch of applied mathematics that revolves around quantifying how much information is present in a signal. It was originally invented to study sending messages from discrete alphabets over a noisy channel, such as communication via radio transmission. [30]

When the dataset is highly unabalanced, i.e. the frequencies of the examples belonging to each class are very unbalanced, *weighted* cross-entropy is used:

$$LWeightedCE = -\sum_{i=1}^{n} w_i(y_i \log(\hat{y_i})) \tag{2.22}$$

where w_i is the weight associated to the ith class. Suppose that the class i have n_i examples belonging to it, the easiest way to calculate the weight w_i is $1/n_i$: the more examples there are in the ith class, the less the weight w_i will be.

2.2.6 Deep Learning

Deep Learning is the most advanced branch of Machine Learning. It is a set of techniques based on artificial neural networks organized in different layers: each layer calculates the values for the next one, in order to process the information in an increasingly complete way. With the term Deep Neural Network (DNN) we denote networks composed of many layers (at least 2 hidden layers, but usually many more) organized hierarchically. Hidden layers of a deep neural network are not all the same, they do different things in different ways: this is an important difference compared to the traditional MLP. In section 2.2.3 we described MLPs and we said that they are universal approximators, i.e. they can approximate any function. Theoretically we could always use a MLP, but in practice this is not possible because the solution may not necessarily be found efficiently. That's why we need deep neural networks: the increase of depth of neural networks and a hierarchical organization of layers made it possible to solve very hard problems.

One of the most significant advantages of a deep neural network is that it is able to learn automatically the best representations of data for solving the problem. This aspect is known as *representation learning* and is in contrast with traditional machine learning algorithms. For example, the *Support Vector Machine (SVM)* [26] is a geometric model which can find an *optimal* decision boundary to separate two linearly separable classes. When facing a non-linearly separable dataset, *SVM* needs a *hand-crafted* transformation function (called *kernel function*) which projects the input space in a new space where examples are linearly separable and *SVM* can solve the problem. Instead, deep neural networks learn to perform such transformations autonomously.

Furthermore, features of data that can be used for solving the problem are automatically learnt by the network. This aspect is called *unsupervised feature learning*. Suppose a face recognition application: the raw input may be a matrix of pixels; the first representational layer may abstract the pixels and encode edges; the second layer may encode a nose and eyes; and the third layer may recognize that the image contains a face. Figure 2.7 gives a graphical intuition of this aspect.

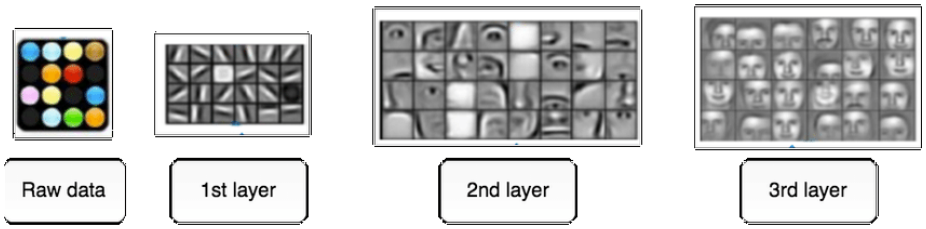


Figure 2.8: A neural network learns features: the first layer learns to recognize edges and shapes; the second one identifies features like nose and eyes; the third one identifies a human face. These are representations obtained with a weighted linear combination of the previous layers filters. Source: [24]

The reasons of Deep Learning success

Clearly, a deep neural network is very harder to train than a traditional MLP: it needs more data and more computational power. The recent success of Deep Learning is fundamentally due to these factors:

- The increase in available data, which have now reached very high quantities.
- The development of high-performance parallel computing systems based on GPU (Graphics Processing Unit).
- The resolution of the *vanishing gradient* problem. Traditional activation functions such as the sigmoid function have a problem which amplifies as the hidden levels increase: at a certain distance from the origin, the function falls flat and therefore the derivative becomes close to zero. The vanishing gradient problem is that the backpropagated gradient becomes too small for upgrading the weights of the neurons. The opposite problem is the *exploding gradient* and occurs when the derivative is too high: the error signal becomes uncontrollable and the weights are updated too "violently". There has been a lot of research works to solve this problem. In fact, particular architectures have been designed that favor the flow of the gradient, such as the ResNet [33]. Furthermore, the ReLU function activation presented in section 2.2.1 (at least partially) solves this problem. It has null derivative for negative values and constant derivative for positive values. An important and widely used variant of the ReLU function is the *Leaky ReLU*: it is the same as ReLU except that it has a line with a negative slope for negative values. Figure 2.9 shows the graph of the Leaky ReLU.

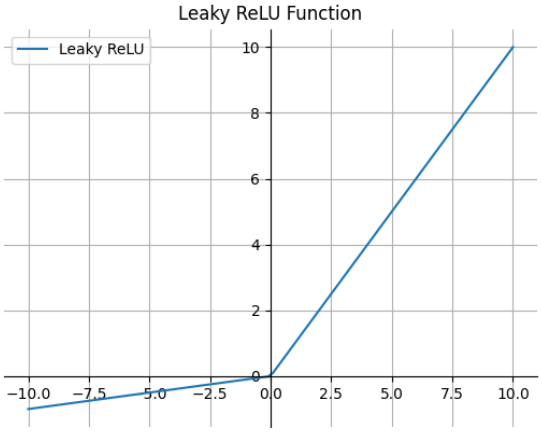


Figure 2.9: Leaky ReLU function

Deep Learning in everyday life

Deep Learning is one of the main sources of success for the field of Artificial Intelligence. Thanks to artificial neural networks, we are able to automatically analyze data such as images, videos, audio or time series. As already mentioned, Machine Learning is present in many aspects of the modern society:

- Image Classification: A common evaluation set for image classification is the MNIST dataset [43]. MNIST is composed of handwritten digits and includes 60,000 training examples and 10,000 test examples (see figure 2.10). State-of-the-art architectures have a >99% accuracy in the MNIST dataset.
- Natural language processing: in this field recurrent neural networks found their success for tasks such as sentiment analysis, information retrieval, machine translation and others.
- Security: for example video surveillance and facial recognition.
- Medical Image Analysis: Deep Learning has been shown to produce competitive results in medical application such as cancer cell classification, lesion detection, organ segmentation.

Figure 2.10: MNIST database. Source: [6]

2.3 State-of-art network architectures

2.3.1 CNN

Convolutional networks (LeCun, 1989) [44], also known as convolutional neural networks (ConvNet) or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. The name indicates that the network employs a mathematical operation called **convolution** which is a specialized kind of linear operation. Convolutional networks have been tremendously successful in practical application in various fields such as image and video recognition, image classification, image segmentation, medical image analysis and natural language processing.

What is a ConvNet?

CNN is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics. The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual

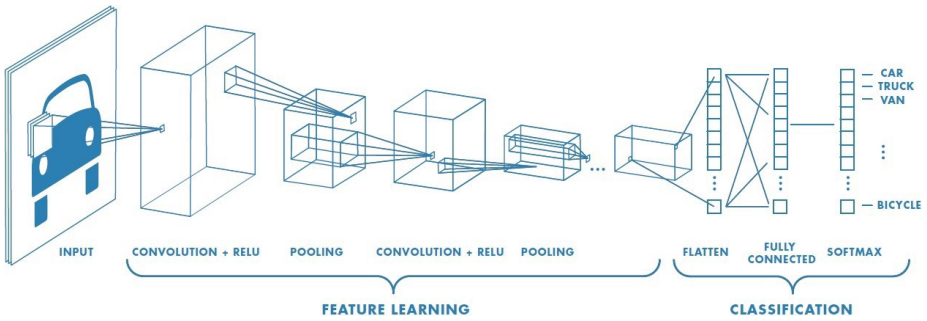


Figure 2.11: Convolutional Neural Network. We can identify the first part in which the features are learned while in the second one the classification is done. Source: [7]

neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

A ConvNet is able to successfully capture the spatial and temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

CNN takes an input image (usually in RGB) and its main role is to reduce the image into a form which is easier to process, without losing features which are critical for getting a good prediction.

Architecture

A ConvNet is similar to a MLP neural networks, but it has some important differences:

- *Local connectivity*: neurons are connected to those of the previous level only locally. This implies that each neuron carries out a local processing, i.e. it sees only a part of the previous level, reducing the connections and specializing the weights.
- *Sharing weights*: Weights are shared so different neurons of the same level process in the same manner different portions of the input resulting in a large reduction in the number of weights used.

CNNs have some distinctive levels, let's examine them briefly. The first is the

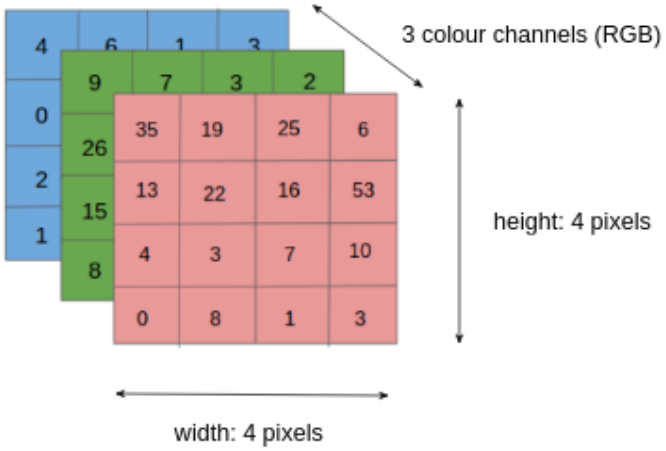


Figure 2.12: An input image. It has a value for each pixel for three dimensions. Source: [7]

Convolution Layer, which apply the convolution operation. The element involved in carrying out this operation is the Kernel: it is a matrix (k) which moves on a portion of the input image (p) every time performing a matrix moltiplication, moving with a certain stride value on the entire image. The aim of the convolution operation is to extract the high-level features such as edges, from the input image. ConvNets need not be limited to only one Convolutional Layer. Conventionally, the first ConvLayer is responsible for capturing the Low-Level features such as edges, color, gradient orientation, etc. With added layers, the architecture adapts to the High-Level features as well, giving us a network which has the wholesome understanding of images in the dataset, similar to how we would.

Then there is the **Pooling layer**, which is similar to the convolution, but this is responsible for reducing the spatial size of the convolved features. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model. There are two types of Pooling: **Max Pooling** and **Average Pooling**. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values from the portion of the image covered by the Kernel.

Afterwards there is the **ReLU Layer**. ReLU is the abbreviation of rectified linear unit, which applies the non-saturating activation function $f(x) = max(0, x)$. It effectively removes negative values from an activation map by setting them

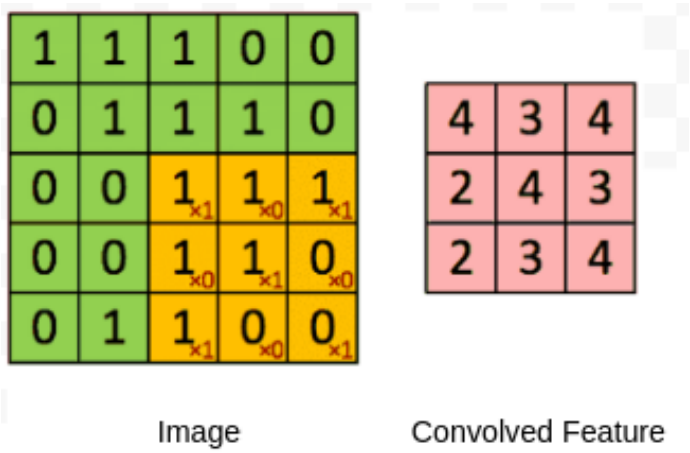


Figure 2.13: An input image and the filter. Passing the kernel over the image and multiplying the matrices we obtain the convolved feature. Source: [7]

to zero. It introduces nonlinearities to the decision function and in the overall network without affecting the receptive fields of the convolution layers.

After several convolutional and max pooling layers, the final classification is done via fully connected layers.

Finally, in the output layer, the **softmax loss function** is used for predicting a single class of K mutually exclusive classes.

2.3.2 ResNet

ResNet [33] was developed in 2015 by the Microsoft Reaserch team. It has achieved great results in several competitions, thanks to its characteristic of allowing much deeper networks than other architectures such as VGG through the introduction of **residual connections**.

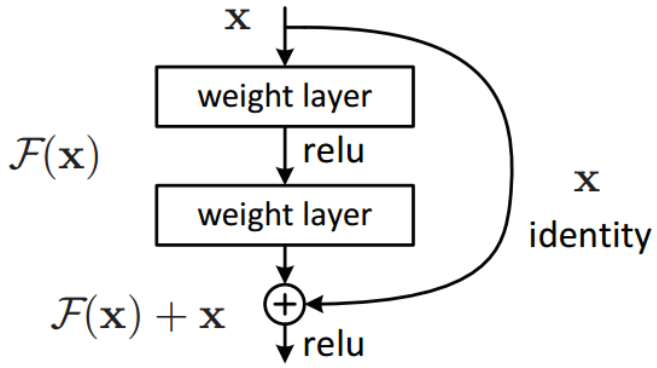


Figure 2.14: Example of residual connection. It is a shortcut from x to the output block. Source: [34] p. 2

In Fig. 2.14 a residual connection is showed: $F(x)$ denotes the output of a block of layers on a certain input x. The residual connection consists in adding a shortcut from x to the block output (also called *skip connection*), which allows the gradients to flow in two directions. The reason for skip connections is that, when dealing with deep networks, gradients can become increasingly small after each layer and this could prevent the network from learning. This problem is known as gradient degradation. Residual connections help in attenuating this problem, by providing an alternative path for gradients flow, without experiencing the degradation problem. A block of layers with a residual connection is called residual block. Residual blocks are repeated multiple times along the depth of the network. The original paper provides different ResNet variants, containing up to 152 layers. Resnet-101 was used in this work as the encoder for our U-Net, whose architecture is described in the next section.

2.3.3 U-NET

U-Net [54] is a convolutional neural network that was developed for biomedical image segmentation at the Computer Science Department of the University of Freiburg (2015).

The typical use of convolutional networks is on classification tasks, where the output to an image is a single class label. However, in many visual tasks, especially in biomedical image processing, the desired output should include localization, i.e. a class label is supposed to be assigned to each pixel.

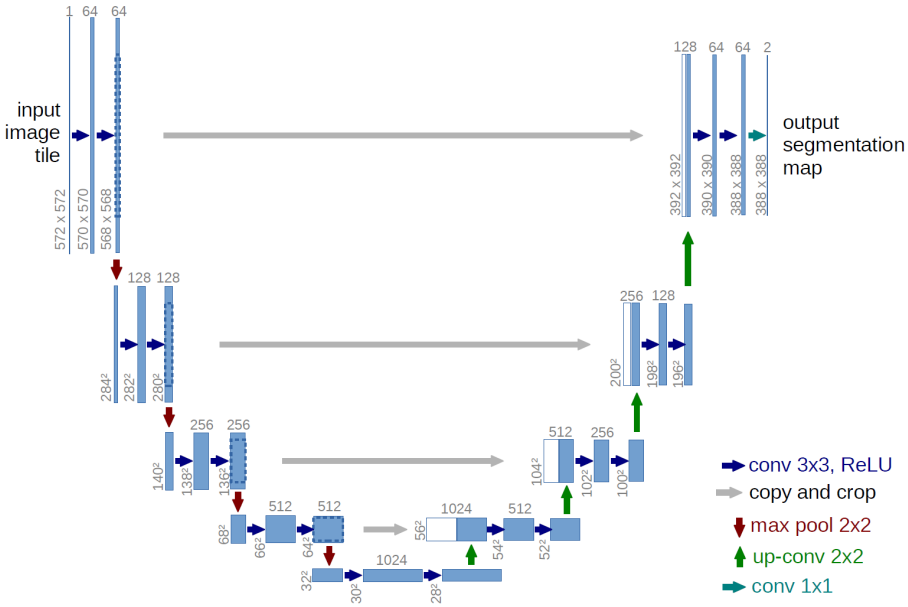


Figure 2.15: U-Net Architecture. The name derives from its particular U-shaped architecture. Source: [54] p.2

Architecture

The network architecture is illustrated in Fig. 2.15 . It consists of a contracting path on the left side (encoder) and an expansive path on the right side (decoder). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU), scheme in fig. 2.16.

Then, there is a 2x2 Max Pooling operation with stride 2 to conclude the downsampling (scheme fig. 2.17). At each downsampling step we double the number of feature channels.

Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 up-convolution that halves the number of the feature channels, a concatenation with the correspondingly cropped feature map from the contracting path and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. In total the network has 23 convolutional layers.

Moreover in the figure we can see some *skip connections* that connect levels from

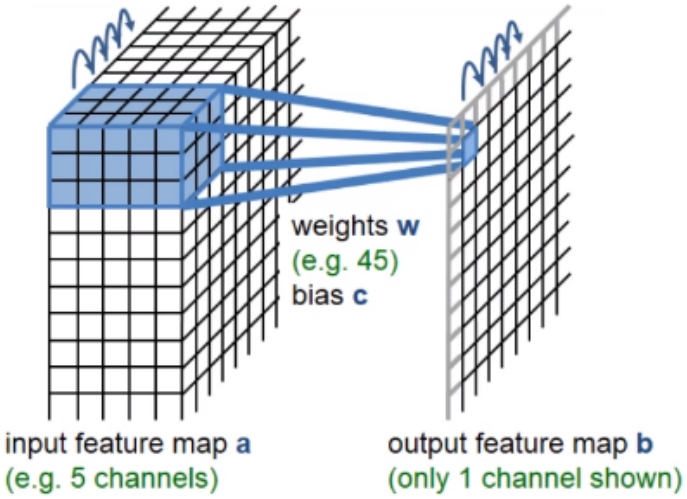


Figure 2.16: The convolution operation scheme. In the U-Net, the convolution with weights w and then the ReLU function are applied to the input feature map. Source: [8]

left to right. Long skip connections are used to pass features from the encoder path to the decoder path in order to recover spatial information lost during downsampling. It basically means that the network first learns *"what"* there is in the image, but it lost the *"where"* information. The latter is recovered by the decoder by gradually applying up-sampling. Then to get better precise locations, at every step of the decoder we use skip connections by concatenating the output of the transposed convolution layers with the feature maps from the Encoder at the same level.

To conclude, image segmentation is an important problem and U-Net contributed significantly in such research. Every day some new research papers are published and many new architectures are inspired by the U-Net.

2.3.4 Generative Adversarial Network (GAN)

A Generative Adversarial Network (GAN) is a class of machine learning frameworks designed by Ian Goodfellow and his colleagues in 2014. Two neural networks, the *discriminator* and the *generator*, contest with each other in a game, where the generator learns to generate new data with the same statistics as the training set. For example, a GAN trained on photographs can generate new photographs that look at least superficially authentic to human observers, having many realistic characteristics.

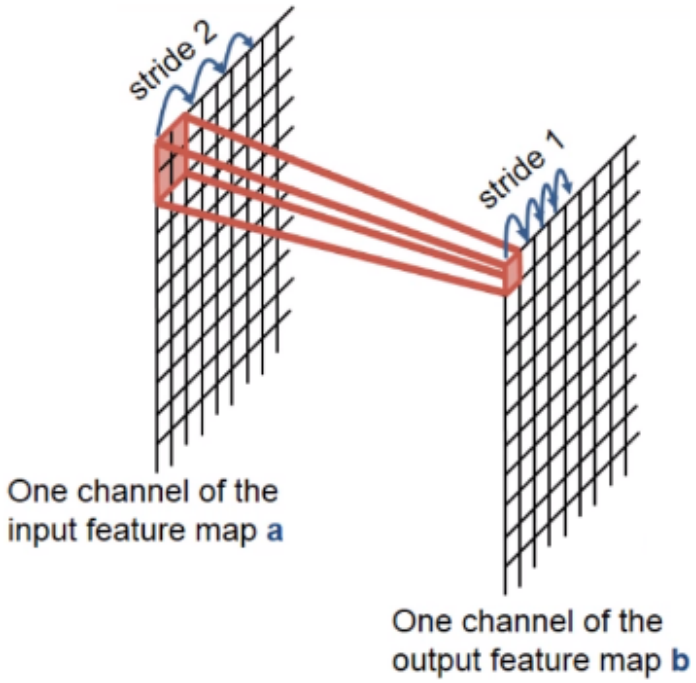


Figure 2.17: MaxPooling function. It takes the bigger value from the input feature map covered by the kernel. With a 2x2 kernel and stride 2 the resulting feature map has factor 2 lower spatial resolution. Source: [8]

But first of all, what does *generative* mean in the name Generative Adversarial Network? "Generative" describes a class of statistical models that contrasts with discriminative models. Let's briefly explore the concept of generative model.

Generative Models

From the point of view of statistical classification, discriminative models model the *posterior probability distribution* $P(Y|X)$, where Y is the target variable and X are the features. That is, given X, they return a probability distribution over Y. A discriminative model models $P(Y|X)$ but not $P(X)$, and hence can be used to label data but not to generate it. So, here the task is to build models able to label unseen examples.

Instead, generative models are generally more complex: they model the joint distribution $P(X, Y)$ of the target Y and the feature vector X. Once we have access to this joint distribution, we can derive any conditional or marginal distribution involving the same variables. In particular, since $P(X) = \sum_y P(Y = y, X)$ it

follows that the posterior distribution can be obtained as:

$$P(Y|X) = \frac{P(Y,X)}{\sum_y P(Y=y,X)} \tag{2.23}$$

Alternatively, generative models can be described by the likelihood function $P(X|Y)$, since $P(Y,X) = P(X|Y)P(Y)$ and the target or prior distribution can be easily estimated or postulated. Such models are called "generative" because we can sample from the joint distribution to obtain new data points together with their labels. Alternatively, we can use $P(Y)$ to sample a class and $P(X|Y)$ to sample an instance for that class. Lastly, note that where there are no labels associated to the data, we are in *unsupervised learning* and we model only the distribution $P(X)$.
Summarizing:

- Generative models capture the joint probability $P(X,Y)$, or just $P(X)$ if there are no labels. They can generate new data instances.
- Discriminative models capture the conditional probability $P(Y|X)$. They discriminate between different kinds of data instances.

Neither kind of model has to return a number representing a probability. You can model the distribution implicitly. For example, a discriminative classifier like a *decision tree* [28] can label an instance without assigning explicitly a probability to that label.

From a certain point of view, discriminative models try to draw boundaries in the data space, while generative models try to model how data is placed throughout the space. For example, the figure 2.18 shows discriminative and generative models of handwritten digits. The discriminative model tries to tell the difference between handwritten 0's and 1's by drawing a line in the data space. If it gets the line right, it can distinguish 0's from 1's without ever having to model exactly where the instances are placed in the data space on either side of the line. In contrast, the generative model tries to produce convincing 1's and 0's by generating digits that fall close to their real counterparts in the data space. It has to model the distribution throughout the data space.
GANs offer an effective way to train such rich models to resemble a real distribution. To understand how they work we'll need to understand the basic structure of a GAN.

GAN structure

The basic idea of GANs is to set up a game between two players:

- The *generator* learns to generate plausible data: it creates samples that are intended to come from the same distribution of the training data.

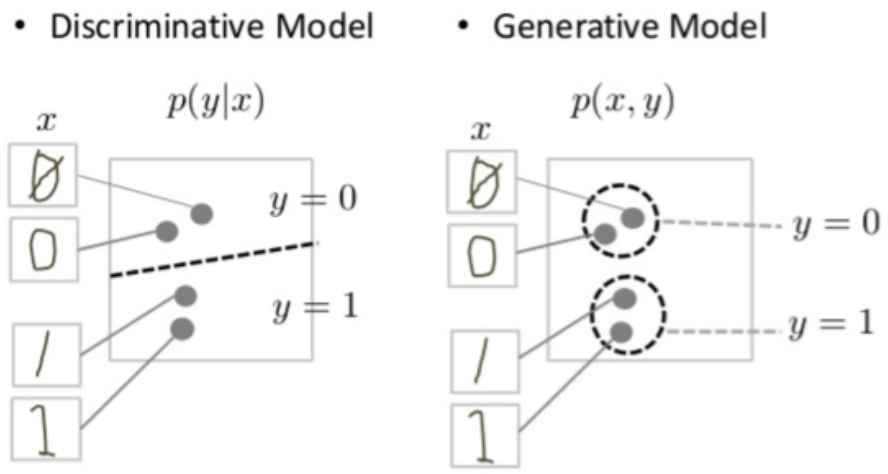


Figure 2.18: Discriminative and generative models of handwritten digits. Source: [9]

- The *discriminator* learns to distinguish the generator's fake data from real data.

Figure 2.19 shows an architecture overview of a GAN.

The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it is classifying. The discriminator's training data comes from two sources:

- Real data instances $\boldsymbol{x} \sim p_{data}$ such as real pictures. The discriminator uses these instances as positive examples during training.
- Fake data instances created by the generator. The discriminator uses these instances as negative examples during training.

Formally, the discriminator is a function D that uses $\boldsymbol{\theta}^{(D)}$ as parameters. During training, the discriminator classifies both real data and fake data from the generator producing an output value $D(\boldsymbol{x}) \in [0, 1]$ (usually the last layer is composed by a neuron with a sigmoid activation function). Real examples are classified with $y = 1$ and fake examples are classified with $y = 0$. The discriminator loss penalizes the discriminator for misclassifying a real instance as fake or a fake instance as real.

The generator is defined by a function G that takes $\boldsymbol{z} \sim p_z$ as input and uses

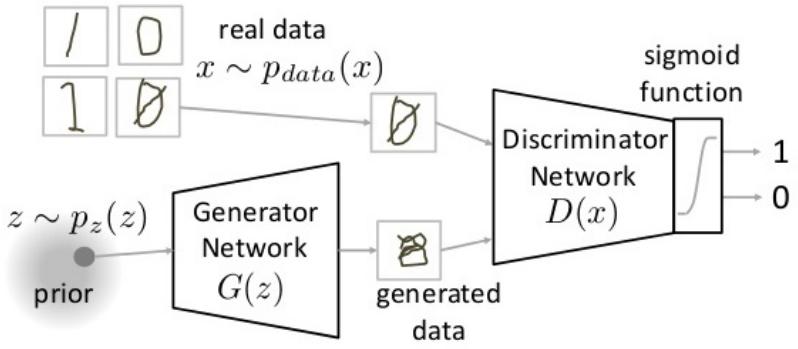


Figure 2.19: Generative Adversarial Network for handwritten digits.

$\boldsymbol{\theta}^{(G)}$ as parameters. The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to trick the discriminator into classifying its output as real. What do we use as input for a network like the generator that outputs entirely new data instances? In its most basic form, a GAN takes random noise $\boldsymbol{z} \sim p_z$ as its input: we can choose something that's easy to sample from, like a uniform or a gaussian distribution. The generator then transforms this noise into a meaningful output. The generator feeds into the discriminator net, and the generator loss (based on the output of the discriminator) penalizes the generator for producing a sample that the discriminator network classifies as fake.

Both players have cost functions that are defined in terms of both players' parameters. The discriminator wishes to minimize $J^{(D)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)})$ and must do so while controlling only $\boldsymbol{\theta}^{(D)}$. The generator wishes to minimize $J^{(G)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)})$ and must do so while controlling only $\boldsymbol{\theta}^{(G)}$. Each player's cost depends on the other player's parameters, but each player cannot control the other player's parameters.

The training process consists of simultaneous SGD. On each step, two minibatches are sampled, one from real data and one from generated data. Then two gradient steps are made simultaneously: one updating $\boldsymbol{\theta}^{(D)}$ to reduce $J^{(D)}$ and one updating $\boldsymbol{\theta}^{(G)}$ to reduce $J^{(G)}$. In both cases, it is possible to use the gradient-based optimization algorithm of your choice. Adam [42] is usually a good choice.

GAN Loss functions

Discriminator
The cost used for the discriminator is:

$$J^{(D)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}) = -\frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim p_{data}} \log D(\boldsymbol{x}) - \frac{1}{2}\mathbb{E}_{\boldsymbol{z}\sim p_z} \log\left(1 - D\left(G(\boldsymbol{z})\right)\right) \quad (2.24)$$

All versions of the GAN game encourage the discriminator to minimize equation 2.24.

This is just the standard binary cross-entropy cost that is minimized when training a standard binary classifier with a sigmoid output:

$$CrossEntropy(y, \hat{y}) = -\mathbb{E}_{\boldsymbol{x}}[y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \quad (2.25)$$

where $\hat{y}$ is the result of the discriminator. In our case, when $y = 1$ we are dealing with *real* examples and the prediction we are interested in is $D(\boldsymbol{x})$ with $\boldsymbol{x} \sim p_{data}$. When $y = 0$ we are dealing with *fake* examples and the prediction is given by $D(G(\boldsymbol{z}))$ with random noise $\boldsymbol{z} \sim p_z$.

Generator
The simplest version of the game between the discriminator and the generator is a *zero-sum game*, in which the sum of all player's costs is always zero. In this version of the game:

$$J^{(G)} = -J^{(D)} \quad (2.26)$$

Because $J^{(G)}$ is tied directly to $J^{(D)}$, we can summarize the entire game with a value function specifying the discriminator's payoff:

$$V(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}) = -J^{(D)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}) \quad (2.27)$$

Zero-sum games are also called minimax games because their solution involves minimization in an outer loop and maximization in an inner loop:

$$\boldsymbol{\theta}^{(G)*} = \arg\min_{\boldsymbol{\theta}^{(G)}} \max_{\boldsymbol{\theta}^{(D)}} V(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}) \quad (2.28)$$

Setting the cost as equation 2.26 for the generator is good only for theoretical analysis, but it does not perform particularly well. In this setting, the discriminator *minimizes* the cross-entropy, but the generator *maximizes* it. This is unfortunate because the gradient of the loss vanishes when the discriminator rejects the generator examples with high confidence. To solve this problem, one can make the generator to minimize a cross-entropy term tailored to its view of the problem:

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_z \log D(G(\boldsymbol{z})) \quad (2.29)$$

This time the two losses are not the one the inverse of the other and the game is no longer a zero-sum game. Equation 2.29 is the loss truly used for the generator when training a GAN.

GAN results

Lots of works using GANs have been proposed in the literature with impressive results. Most GANs today are at least loosely based on the DCGAN architecture [52]. Principal features of DCGANs are:

- Replace any deterministic spatial pooling layers with strided convolutions, allowing the network to learn its own spatial downsampling.
- Use BatchNorm [37] in both the generator and the discriminator.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh. Use LeakyReLU activation in the discriminator for all layers.
- Use Adam optimizer rather then SGD with momentum.

A noteworthy work is the StyleGAN [40] developed by NVIDIA: it is a GAN able to create very detailed and high resolution images. Unbelievable results of human faces generated (and therefore that they do not exist!) by a StyleGAN are shown in figure 2.20

Figure 2.20: Human Faces generated by a StyleGAN. You can view other generated examples at `https://thispersondoesnotexist.com`

Chapter 3

Datasets

The computational pathology (CPath) is an emerging area for the application of deep learning methods in medical imaging tasks: different studies have demonstrated the potential of Deep Learning models in detecting cancer, classifying tissue, identifying diagnostically relevant structures and even inferring genetic sub-types [49] [25] [56]. However, an important point to remember is that the great success of algorithm in computer vision and deep learning applied to natural images and medical imaging tasks can be attributed to the availability of large datasets.

In this chapter we introduce the datasets that were used in this work. We employed two different datasets for a variety of tasks: PanNuke and UniToPatho.

3.1 PanNuke

PanNuke [29] is the largest and most assorted dataset for the segmentation and classification of cells nuclei in the computional pathology field. The data was collected from a series of existing datasets and then improved. It was semi-automatically annotated and qualitatively checked by clinical professionals. The great advantage of this dataset is that models that use it for training are able to generalize even on tissues such as the brain that are not part of the dataset.

This work was motivated by the fact that publicly available nucleus segmentation and classification datasets do not often match the distribution of data in the clinical field, so it can be observed that when the model is applied to a few images that contain commonly found artifacts in clinical practice, there are several false detections which could lead to incorrect or misleading results.

In our work, PanNuke was used to train a U-Net able to perform the semantic segmentation of histopathological images, according to the different cell types. Once the U-Net has been trained, it was used to produce the semantic segmentation masks of UniToPatho and, at the end, the segmentation masks produced were used to train our GAN, in order to translate the masks into realistic histopathological images.

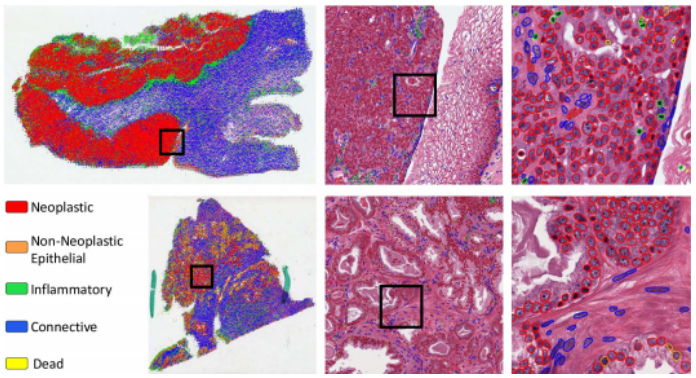


Figure 3.1: Images of PanNuke. We can see the segmentation and classification of cells of 5 different classes in the tissues. Source: [29]

Dataset Generation

PanNuke has been aggregated from a set of publicly available nucleus classification and detection datasets for creating initially a dataset for semi-automatic ground truth generation. Then, using a Fully Convolutional Neural Network (FCNN) trained on the extracted dataset, which was previously re-labeled according to the PanNuke categories, all the detected nuclei have been classified and finally verified by a team of expert pathologists. All the whole-slide images of the dataset have been scanned with a maximum resolution of either 20x or 40x, but then re-sized all to 40x. Furthermore all images have been split into patch of 256 x 256 pixel dimension. For the last version of PanNuke, *NuClick* [39] has been used for producing accurate segmentation masks from single points. Therefore using the proposed semi-automatic pipeline, there have been generated and verified 189,744 nuclei from more than 20,000 WSIs.

Dataset Schema

PanNuke contains data from 19 different tissues like:

- Breast

- Colon
- Lung
- Bladder
- Liver
- Stomach, etc.

Thus the nuclei classification is shared across them. Cells, then, are split between two macro areas: *Neoplastic* and *Non-neoplastic* cells. Neoplasm includes any tumor, malignant or benign. It includes carcinomas, sarcomas, melanomas, lymphomas, etc. These are all tumors but originate from different cell types: carcinomas from epithelial; sarcomas from soft tissue; melanomas from melanocytes; lymphomas from lymphoid cells and so on. As such, all tumorous cells in PanNuke are labeled as Neoplastic. Non-neoplastic covers everything else, from normal to inflammatory, degenerative, metaplastic, atypia etc. As such, non-neoplastic labels in PanNuke are:

- Epithelial
- Connective/soft tissue cells
- Inflammatory
- Dead cells

Here, connective tissue cells have the potential to become neoplastic, whereas inflammatory cells typically cannot become neoplastic. Inflammatory cells include lymphoid and macrophage cells in PanNuke. Dead cells can arise from either neoplastic or non-neoplastic cells, but in this dataset it is referred as non-neoplastic. Figure 3.2 shows an example of an image and its segmentation mask. To draw the mask, a color was used for each label, as follows:

- Neoplastic - blue
- Inflammatory - green
- Connective/Soft tissue - red
- Dead - light blue
- Epithelial - violet

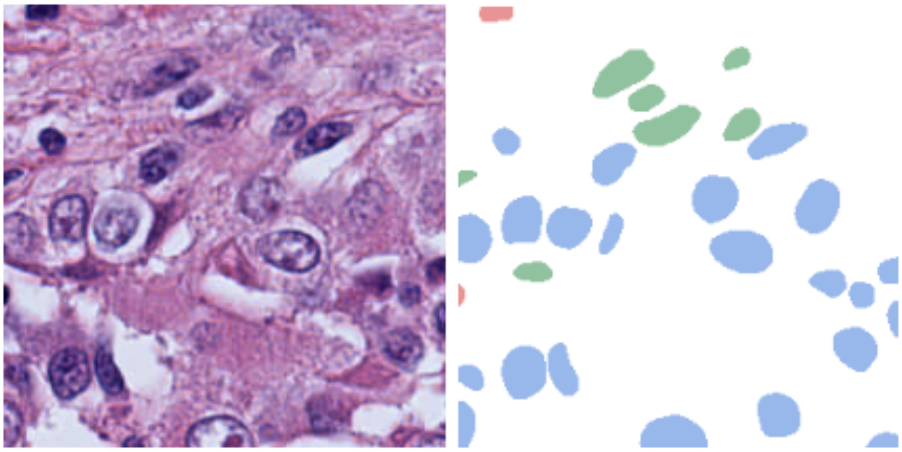


Figure 3.2: An image of PanNuke (left) with the segmentation mask (right). Cells of different types are classified with different colours.

Statistics

For each tissue, the number of cells of each type changes. During the annotation, pathologists concentrated mainly on observing the surrounding structures of each cell. Instead, *Sirinunkunwattana et al.* 's [57] automatic nucleus classification system only focuses on small patches of images containing one or only a few cell nuclei.

However, these two different approaches have good average accuracy. Recent studies by Oakden et al. [51] demonstrated the effects of 'hidden stratifications' in medical images, whereby the labels used in CV and ML do not represent what is reality making average performance not a good measure of applicability. Due to this stratification phenomenon there are a series of cells limited to some tissue regions that can hardly be classified correctly by any algorithm. It is important to say that the distinction between neoplastic and non-neoplastic cells **is not an easy challenge even for pathologists**, who, before being able to give a result, need contextual information.

3.2 UniToPatho

UniToPatho [20] is a dataset of annotated high-resolution of Hematoxylin and Eosin (H&E)-stained colorectal images, comprising different histological samples of colorectal polyps, collected from patients undergoing cancer screening. Colorectal polyps characterization relies on the histological analysis of tissue

samples to determine the polyps malignancy and dysplasia grade. Also this allows to tailor patients management and follow up with the ultimate aim of avoiding or promptly detecting an invasive carcinoma.

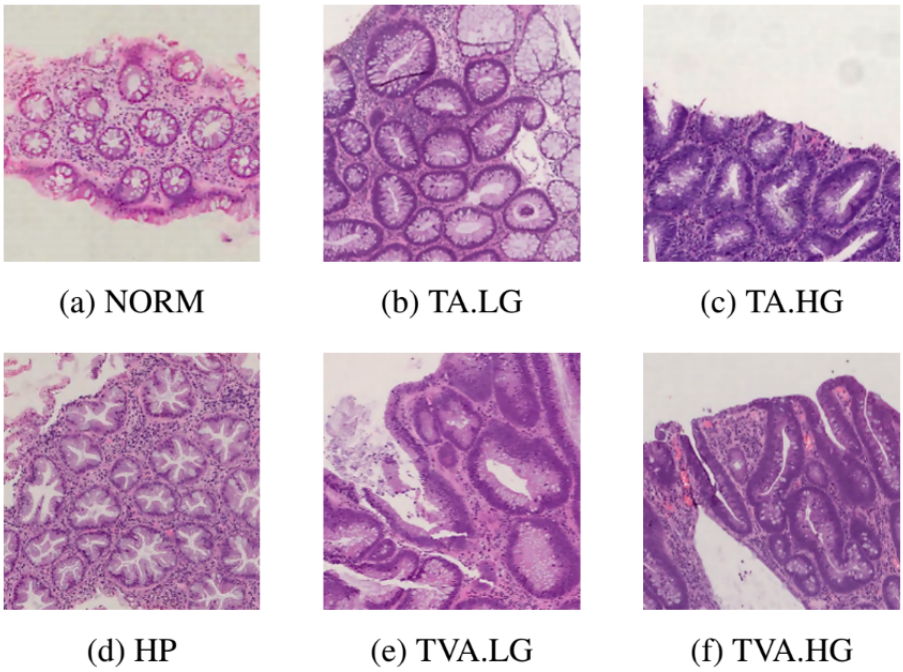


Figure 3.3: UniToPatho classification: there are six different classes in which the tissue is classified. Source [20]

Gastrointestinal histopatolohists inspect tissue samples collected during colonoscopies, looking for hints that can predict the insurgence of invasive carcinoma [23]. Colorectal polyps are pre-malignant lesions found in the instasinal mucosa that pathologists analyze to *i)* ascertain the polyp type (hyperplastic, adenoma) and *ii)* assess the dysplasia grade in case of adenomas.

The UniToPatho is a collection of the most relevant patch images extracted from 292 whole-slide images that are acquired at 20x magnification. Each slide has been cropped with multiple non-overlapping square patches at different scales. Denoting the side of the physical area of a patch with σ, measured in μm , in the dataset we find 8669 images of which extracted at $\sigma = 800$ (1812×1812 pixels patches) and 867 at $\sigma = 7000$ (15,855×15,855 pixels patches).

Each slide belongs to a different patient and is annotated by expert UniTo pathologists, according to six classes as follows:

- NORM - Normal tissue

- HP - Hyperplastic Polyp
- TA.HG - Tubular Adenoma, High-Grade dysplasia
- TA.LG - Tubular Adenoma, Low-Grade dysplasia
- TVA.HG - Tubulo-Villous Adenoma, High-Grade dysplasia
- TVA.LG - Tubulo-Villous Adenoma, Low-Grade dysplasia

Hyperplastic polyps usually exhibit no malignant potential [60], while adenomas are more likely to progress into invasive carcinomas. Tubular and tubulo-villous are common colorectal adenomas, with villous adenomas generally presenting higher malignant potential given the larger surface. Adenomas are associated with a grade of dysplasia, which measures the abnormality in cellular growth and differentiation. **Higher grade dysplasia indicates higher malignant potential.**

Chapter 4

Segmentation

In this chapter we will describe briefly how we dealt with the problem of generating semantic segmentation masks of histopathological images. Personally, I worked both for segmentation and generation task (with my colleague Davide Di Luccio), but this work only focuses on the generation task. What will be described below in this chapter is only a summary of the work done and all the details on the segmentation task can be found in [45], which is my colleague Davide Di Luccio's thesis.

4.1 The semantic segmentation task

What is semantic segmentation? The goal of semantic image segmentation is to label each pixel of an image with a corresponding class of what is being represented. Because we are predicting the label for every pixel in the image, this task is commonly referred to as dense prediction. The output is a high resolution image (typically of the same size as input image) in which each pixel is classified to a particular class. Thus it is a pixel level image classification. This task is different to the *instance segmentation task* where models distinguish between separate objects of the same class. In other words, if you have two objects of the same category in your input image, the segmentation map does not inherently distinguish these as separate objects.

The goal is to take either a RGB color image (height x width x 3) or a grayscale image (height x width x 1) and its output segmentation map where each pixel contains a class label represented as an integer. The target is created by one-hot encoding the class labels, using an output channel for each of the possible classes (height x width x classes). Thus, a prediction can be also collapsed into

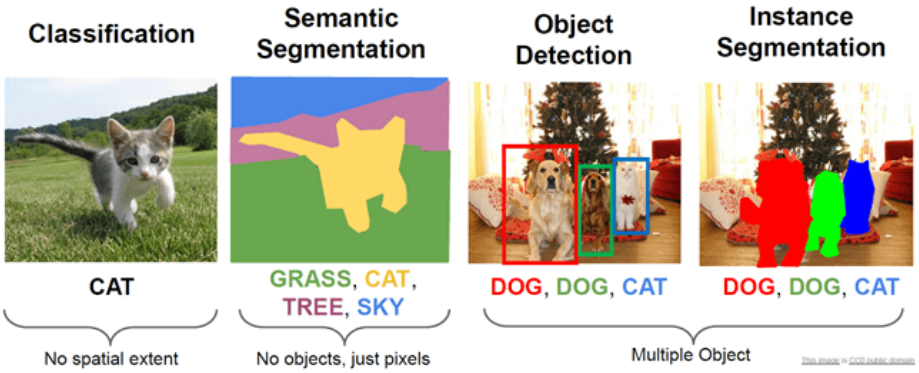


Figure 4.1: Different tasks on images: the classification says what there is in the image; the semantic segmentation divides all differents objects; object detection detect each object; instance segmentation identifies each instance of objects. Source: [10]

a segmentation map by taking the *argmax* of each depth-wise vector (height x width). To be clearer, we show the code which transforms a one-hot encoded mask into a mask where the pixels are labeled with integers, using the *argmax* function.

```
import torch
one_hot_mask = < mask of shape (CLASSES, H, W) >
# call the argmax() function across the classes dimension
integer_mask = torch.argmax(one_hot_mask, dim=0)  # shape: (H, W)
```

Listing 4.1: Code used to transform a one-hot encoded mask in mask where pixels are labeled with integers.

Figure 4.2: Road scene image segmentation. Source: [11]

4.2 Training

In this phase we trained the U-Net for the semantic segmentation task using the PanNuke dataset. In fact, we wanted to have a network so that it could be used to generate the segmentation masks of the images of the UniToPatho dataset. The training of this neural network was not very easy, first of all given the difficult nature of the task and then by the particularities of the U-Net that made it complex in some aspects.

PanNuke contains images which are similar to those of our UniToPatho but, most importantly, it is labeled, i.e. for each image the relative segmentation mask is given.

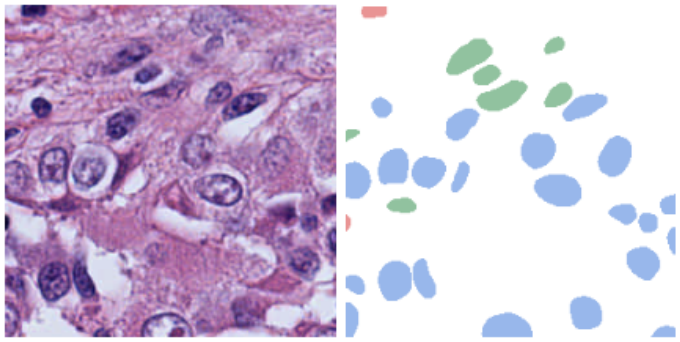


Figure 4.3: An image of PanNuke (left) with the segmentation mask (right). Cells of different types are classified with different colours.

For our experiments the dataset was divided into 3 parts: 70% training set, 15% validation set and 15% test set. All the tests were carried out only on the first two sets of data, while the test set was reserved only for the evaluation of the final network model that we used for UniToPatho. In the first training tests we used the entire dataset, that is with the images belonging to the 19 different types of PanNuke fabrics. Subsequently, however, after a meeting with the pathologists we were advised to use only the images of the colon, because cells of the same type are different in different tissues. In this way the amount of images available for the training was reduced to about a thousand but this did not compromise the final result.

The transformations to be applied to the dataset can be very useful both to modify the images in order to adapt them to the task and to apply data augmentation in order to improve training performance. We started using the transformation of Pytorch (Torchvision), but shortly after we ran into the problem of having to apply the same transformation to both an image and its mask. Therefore we adopted the *Albumentations* [12] library which allowed us to directly manage both the transformations for the images and for the respective

masks. The transformations to be applied to the images were chosen to make sure to preserve the natural appearance of the cells. Therefore, always under the advice of pathologists, we selected only a few transformations that were right for us: Flip (both horizontal and vertical) and RandomRotate90. These data augmentation techniques have only been applied to the training set.

In our work, image were normalized, because normalizing the data generally speeds up learning and leads to faster convergence when training deep neural networks. Since we used a pre-trained network on ImageNet for the encoder of our U-Net, we normalized the images with the *mean* and *standard deviation* of ImageNet:

- mean = [0.485, 0.456, 0.406]
- std = [0.229, 0.224, 0.225]

Several loss functions were studied to train the neural network: (weighted) cross-entropy, Dice loss, generalized Dice loss [58], etc. However, the best results were obtained with a variation of the Dice loss (suggested by Carlo Alberto Barbano) which we named "Mean Dice Loss": it calculates the Dice loss individually for each channel and then returns the mean. The loss is defined as:

$$L_{MeanDiceLoss} = \frac{\sum_{c\ in\ Classes}(1 - Dice\ Score_c)}{|Classes|} \tag{4.1}$$

In order to evaluate our models, we used some important metrics: accuracy, intersection-over-union (IoU), F1 score, precision, recall. However, the metric on which we most focused was the Mean Intersection-Over-Union (mean IoU), which first computes the IOU for each semantic class and then computes the average over classes.

Regarding the architecture of the model, in the introduction chapter we described the U-Net architecture, one of the works that most inspired the construction of semantic segmentation deep learning models architectures. In our work we first tried U-Net, but soon we got better results with a simple variant of it: the U-Net ++ [64]. In U-Net, the feature maps of the encoder are directly received in the decoder; however, in U-Net++, they undergo a dense convolution block whose number of convolution layers depends on the pyramid level. Figure 4.4 shows a simple example of a U-Net++.

4.2.1 Training

Searching for the right network settings has been simplified thanks to the use of the *Sweep* tool from Weights & Biases (wandb) [13].

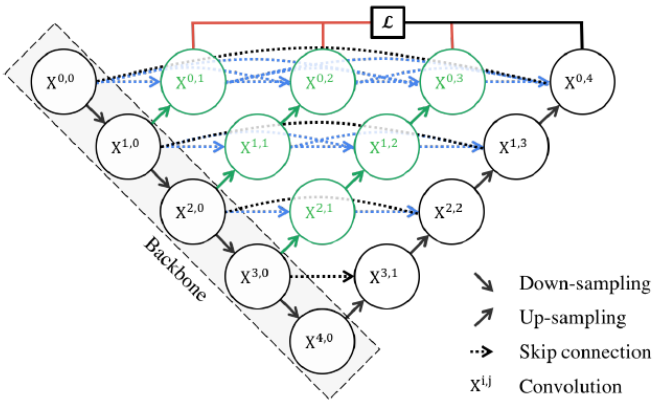


Figure 4.4: U-Net++ architecture. Compared to the original version, this includes a series of nested dense convolutional blocks. In the graphical abstract, black indicates the original U-Net, green and blue show dense convolution blocks on the skip pathways, and red indicates deep supervision. Red, green, and blue components distinguish UNet++ from U-Net. Source: [64]

The best training was done with 40 epochs using an initial *learning rate* of $2 * 10^{-4}$. The learning rate has been modified during the training with the learning rate scheduler *ReduceLROnPlateau* of Pytorch, which was set to divide the learning rate by a factor of 10 after that the *mean IoU* metric stopped improving, with a patience of 8 epochs. Adam optimizer was used with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Weight decay regularization was set to $6 * 10^{-7}$. We used a neural network model offered by the "Segmentation Models Pytorch" (SMP) [62] library which is an excellent resource that provides all the essentials for the semantic segmentation task. In our case we used a U-Net++ with a ResNet-101 encoder pre-trained on ImageNet. We also used the `BatchNorm` layer in the decoder, thanks to the `"decoder_use_batchnorm"` parameter of SMP. The training was done using a Tesla T4-16GB and it took about 1 hour.

4.3 UniToPatho segmentation

Once we had a performing model capable of segmenting PanNuke, our task was to use the network to create the segmentation masks of UniToPatho by making inference.

A first detail is that PanNuke is a dataset whose images are 256x256 pixels, while those of UniToPatho have size of 1812x1812 pixels. In order to supply an image to the network, due to the physical characteristics of the U-Net, this must have a size equal to a power of 2. Therefore, padding has been applied to

the UniToPatho images so as to render the images to the power size of 2 closest to 1812, so 2048x2048 pixels.

Moreover, the PanNuke images have a 40x physical zoom, unlike the UniToPatho ones which are 20x. We decided to apply a resize of the UniToPatho images, that is we doubled the size so as to simulate the zoom.

So, by combining padding and resize, this is the transformation done to UniToPatho images:

```
T = albumentations.Compose([
        albumentations.PadIfNeeded(min_height=2048, min_width=2048),
        albumentations.Resize(height=2048*2, width=2048*2),
        albumentations.Normalize(mean=mean, std=std),
        ToTensorV2()
    ])
```

At this point, the error *"CUDA: out of memory"* occurred: the GPU memory was unable to store the large size of the images to be processed, even using a batch of 1.

To solve this problem, after several tests, we decided to use the *unfolding* technique [14]. Very simply, this technique allows to divide an image into different portions. So, each UniToPatho image of 4096x4096 pixel size was divided into smaller patches of 2048x2048 pixels and then each patch was segmented. At the end, the reverse operation of the unfold technique, the *fold*, was applied, in order to recompose all the various patches and obtain the complete mask.

Furthermore, the coloring of the UniToPatho images is slightly different from that of the PanNuke images on which the model has been trained, due to biomedical reasons related to the creation of WSIs. So we used the *Torchstain* tool [15] for normalizing the histopathological images. This tool allowed us to modify the UniToPatho images to make the coloring more similar to those of PanNuke, thus allowing better performances (since the network was trained on PanNuke).

Finally, with these strategies, the semantic segmentation masks of the entire UniToPatho dataset were created.

Chapter 5

Generation

In this chapter we propose a system to cope with the problem of generating realistic images of histopathological tissue starting from the semantic segmentation masks. Many tasks in computer vision concern with translating an input image into a corresponding output image: for example, coloring a black and white image into a corresponding RGB image, or transforming sketches into photos (see figure 5.1). In order to create images with neural networks, the simplest thing to

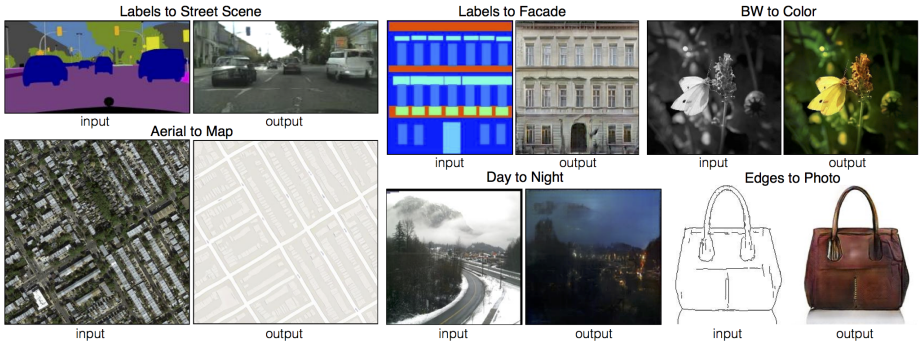


Figure 5.1: Some of image-to-image translation examples. Source: [38]

.

do would be to use a hand-engineered loss function with an autoencoder shaped convolutional neural network. Nevertheless, lots of works in computer vision show that typical loss functions such as MSE, L1, do not produce really good results, as the produced images are often blurry and with unpleasant artifacts. GANs are state-of-the-art models that are capable of producing realistic images. GANs are based on a competition between a discriminator network which tries to distinguish between generated images and real ones, and a generator network

tries to fool the discriminator. Blurry images will not be tolerated since they look obviously fake, and the discriminator network would easily recognize them as such.

5.1 Pix2Pix GAN

5.1.1 Objective

In this section we will describe the structure of a *conditional* generative adversarial network (cGAN) called pix2pix that learns a mapping from input images to output images, as described in [38].

Pix2pix GANs are a variant of the typical GAN: the *conditional* GAN (cGAN) [48]. The loss a classical GAN can be formalized as follows:

$$\mathcal{L}_{GAN}(G, D) = \mathbb{E}_{x \sim p_{data}}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))] \tag{5.1}$$

Generative adversarial networks can be extended to a conditional model if both the generator and the discriminator are conditioned on some extra information y:

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{z,y}[\log(1 - D(y, G(z, y)))] \tag{5.2}$$

where y could be any kind of auxiliary information, such as class labels or data from other modalities. We can perform the conditioning by feeding y into both the discriminator and generator as an additional input layer. In our pix2pix GAN, the discriminator processes both real/fake image and the corresponding segmentation mask (image and mask are concatenated along the channel dimension): clearly the mask represents the additional information provided. Figure 5.2 provides a representation of how the discriminator works to classify real/fake images. While the generator, in order to calculate its loss, will observe not only the segmentation mask from which it must start to produce its output, but also the real image.
Previous approaches have found it beneficial to mix the GAN objective with a more traditional loss, such as L2 distance. The generator is then tasked not to only fool the discriminator but also to be near the ground truth output in an L2 sense. However, since L2 distance seems to produce blurred results, L1 distance was preferred:

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[||x - G(z, y)||_1] \tag{5.3}$$

and an L1 loss between the real image and the fake image is added to the generator loss.

So, the final objective is:

$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G) \tag{5.4}$$

Note that, as suggested in the original GAN paper, rather than training G to minimize $\log(1 - D(y, G(z, y)))$, G is trained to maximize $\log(D(y, G(z, y)))$. In addition, the objective is divided by 2 while optimizing D, which slows down the rate at which D learns relative to G.

Summarizing:

- The discriminator loss is:

$$\begin{aligned} D_{loss} &= -\frac{1}{2}(\mathcal{L}_{cGAN}(D)) \\ &= -\frac{1}{2}(\mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{z,y}[\log(1 - D(y, G(z, y)))]) \end{aligned} \tag{5.5}$$

  which is the traditional GAN loss in the conditional variant, divided by 2.

- The generator loss is:

$$\begin{aligned} G_{loss} &= \mathbb{E}_{x,y,z}[-\log(D(y, G(z, y))) + \lambda \mathcal{L}_{L1}(G)] \\ &= \mathbb{E}_{x,y,z}[-\log(D(y, G(z, y))) + \lambda ||x - G(z, y)||_1] \end{aligned} \tag{5.6}$$

  which is the traditional GAN loss in the conditional variant, plus a L1 loss between the fake image and the real image.

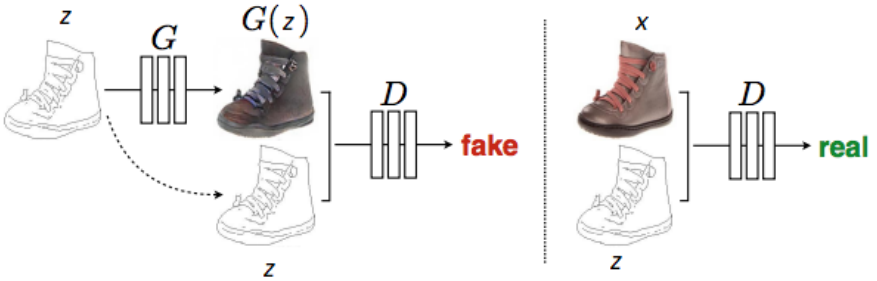


Figure 5.2: Training a conditional GAN to map edges to photo. The generator tries to fool the discriminator learning how to produce nice images starting from the input edge map. The discriminator tries to classify between fake and real images. Both discriminator and generator observe the input edge map. Source: [38]

.

5.1.2 Discriminator

The discriminator is a simple convolutional classifier, composed of some blocks of the form Convolution - BatchNorm - LeakyReLU. The last layer is a convolutional layer with 1 filter followed by a sigmoid activation function.

The peculiarity of the discriminator in the Pix2pix GAN is that it is a convolutional *PatchGAN* classifier, i.e. it tries to classify if each image patch is real or not real. Consequentially, the output of the discriminator is not simply a scalar in (0, 1), but the shape of the output after the last layer is (B, 1, N, N), where each element is in range (0, 1). Thus, this discriminator tries to classify if each N×N patch in an image is real or fake. This is advantageous because a PatchGAN has fewer parameters, runs faster, and can be applied to arbitrarily large images. In order to compute the loss of the discriminator:

- the binary cross-entropy loss of *real* images and an array of *ones* is calculated.
- the binary cross-entropy loss of *fake* images and an array of *zeros* is calculated.
- the final loss is the sum of the previous two ones, divided by 2.

Below a sketch of the code used to calculate this loss:

Listing 5.1: Sketch of the code used to calculate the discriminator loss.

```
real_image = < real image >
mask = < segmentation mask of real_image >
image_fake =  < output of the generator >

# real batch
d_real = disc(mask, real_image)  # output of the discriminator
target = torch.ones_like(d_real)  # tensor of ones
d_real_loss = bce(d_real, target)  # binary cross entropy for reals

# fake batch
d_fake = disc(mask, image_fake)  # output of the discriminator
target = torch.zeros_like(d_fake)  # tensor of zeros
d_fake_loss = bce(d_fake, target)  # binary cross entropy for fakes

d_loss = (d_real_loss + d_fake_loss) / 2
```

5.1.3 Generator

The generator of a Pix2pix GAN is a modified U-Net. It could be described in general terms as a mix between a DCGan and a U-Net. It presents an encoder-decoder architecture with skip connections between mirrored layers in the encoder and decoder stacks. Each block in the encoder is a Convolution - BachNorm - LeakyReLU block and each block in the decoder is of the form TransposedConvolution - BatchNorm - ReLU. Skip connections between each

layer i and layer $n - i$ are added, where n is the total number of layers. Each skip connection simply concatenates all channels at layer i with those at layer $n - i$. The last layer is a convolutional layer with 3 filters (RGB), followed by a tanh activation function.

Similarly to what has been done for the discriminator, the generator loss is a binary cross-entropy loss between the result of the discriminator with the fake images and an array of *ones*. One might ask why the hell we use ones for fake images. The reason is very simple. The generator is trained to maximize $\log(D)$, i.e. to minimize $-\log(D)$. If you look to the formulation 2.25 of the binary cross-entropy considering the label for $y = 1$, you will note immediately that we are dealing with $-y\log(\hat{y})$, i.e. $-\log(D)$.
When the $\mathcal{L}_{L1}(G)$ loss is added to the final loss, it is multiplied by a hyperparameter `L1_LAMBDA`, usually set to 100.

Below a sketch of the code used to calculate the loss of the generator:

Listing 5.2: Sketch of the code used to calculate the generator loss.

```
mask = < segmentation mask >
image_fake = < output of the generator >
d_fake = disc(mask, image_fake)  # output of the discriminator
g_fake_loss = bce(d_fake, torch.ones_like(d_fake))  # binary cross entropy
L1 = L1_loss(image_fake, real_image) * L1_LAMBDA
g_loss = g_fake_loss + L1
```

5.2 Training

5.2.1 Training on PanNuke

At first we tried to train the Pix2pix GAN with the PanNuke dataset, since we had the ground truth masks, while for UnitoPaTho we did not. As explained in section 3.1, PanNuke is a dataset for the segmentation and classification of cells nuclei. It provides data from 19 different tissues like breast, stomach, lungs, etc... Since UniToPatho contains only images of the colon tissue, we trained our neural network using only colon images of PanNuke. Figure 5.3 shows an example of an image with its own segmentation mask.

Dataset Split

Usually, when training a classical GAN, the test set is usually not needed because we can understand if the GAN has learnt well the distribution of data when it

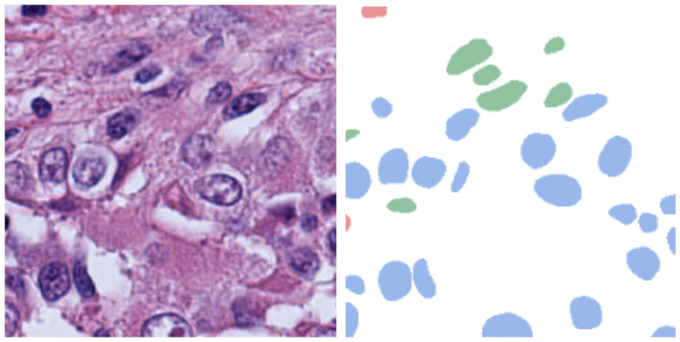


Figure 5.3: An image of PanNuke and its own segmentation mask. Each color corresponds to a type of cell. Blue indicates neoplastic cells, green indicates inflammatory cells and red connective cells. White is used for the background.

starts to produce examples which are very similar to those of the dataset. For our task, instead, we thought that a test set is quite important: we want to see how the GAN behaves when it creates an image for a mask never seen before. Then, it is very interesting to compare the real image with the generated one.

For our experiments the dataset was divided into 2 parts: 90% training set and 10% test set. Test set is so small because we did not compute any evaluation metrics, but we only saw if the generated images were similar to the real ones. Moreover, since we used only images of PanNuke of the colon tissue, the dataset did not contain a lot of images (we had about 1100 available images), so limiting the training set too much made no sense.

Data Augmentation

Since the dataset at our disposal was rather small, we needed to do data augmentation. Note that in our particular context, one have to make the transformations on both the image and the mask. For example, if we rotate an image of 90 degrees, we have also to rotate the segmentation mask of 90 degrees in the same direction. In order to solve this problem there were 2 alternatives. The first one was to use *Torchvision* [16], the package of PyTorch which provides common image transformations for computer vision, and code our transformations appropriately so that they were applied to both the image and the mask in the same way. This can be sometimes very tricky and lengthy to code.

Hence, we decided to use *Albumentations* [12], an image augmentation library which provides easy to use APIs to solve the mentioned problem. The library is widely used in industry, deep learning research, machine learning competitions and open source projects. Albumentations is written in Python and it works

with NumPy [17] arrays, which makes it a very flexible library. For our purposes, it has been very useful because it provides support for augmenting a dataset for the semantic segmentation task: it offers a large variety of transformations applicable to both image and mask.

We decided to apply transformations that did not compromise too much the features of the cells: we used Flip (both horizontal and vertical) and RandomRotate90. At last, images were normalized in the range $[-1, 1]$.

GAN Architecture

We used the same network architectures described at section 6 of [38]. Let `Ck` denote a Convolution-BatchNorm-ReLU layer with k convolutional filters and `CDk` a Convolution-BatchNorm-Dropout-ReLU layer with k convolutional filters and a dropout rate of 50%. All convolutions are 4×4 spatial filters applied with stride 2. Convolutions in the encoder, and in the discriminator, downsample by a factor of 2, whereas in the decoder they upsample by a factor of 2.

The generator has a U-Net architecture, consisting of:

- **encoder**: `C64-C128-C256-C512-C512-C512-C512-C512`
- **decoder**: `CD512-CD512-CD512-CD512-CD512-C256-C128-C64`

BatchNorm is not applied to the first `C64` layer in the encoder. The *input* channels in the decoder are double of those of the encoder because of the skip connections, which concatenate (along the channel dimension) each layer i in the encoder with layer $n - i$ in the decoder, where n is the total number of layers. After the last layer, a convolution is applied with 3 filters (RGB) with a tanh activation function. All ReLUs in the encoder are leaky, while ReLUs in the decoder are not leaky.

The discriminator architecture is: `C64-C128-C256-C512`. BatchNorm is not applied to the first `C64` layer. After the last layer, a convolution is applied with 1 output channel, followed by the sigmoid function. Actually, we omitted the sigmoid function in the discriminator architecture since it is embedded in the loss *BCEWithLogitsLoss* [18] of PyTorch. All ReLUs are leaky.

All LeakyReLUs have a negative slope of 0.2.

Tips and tricks to train a GAN

Training a GAN can be very hard. GANs are sometimes very unstable objects and they often require a lot of trials to find the correct setting and the correct

combination of the hyperparameters. We used some "tricks", taken from [31] and `https://github.com/soumith/ganhacks`:

- Normalization: images were normalized between -1 and 1. Tanh activation function is used at the last layer of the generator.
- Different mini-batches for real and fake were constructed, i.e. each mini-batch needs to contain only real images or only generated images.
- Adam optimizer was used instead of SGD with momentum.
- Label smoothing: instead of using labels as real=1 and fake=0, we replaced the label 1 with a random number between 0.7 and 1.2, and the label 0 with a random number between 0.0 and 0.3.

```
# smoothing class=1 to [0.7, 1.2]
def smooth_positive_labels(label):
    return label - 0.3 + (torch.rand_like(label) * 0.5)

# smoothing class=0 to [0.0, 0.3]
def smooth_negative_labels(label):
    return label + torch.rand_like(label) * 0.3
```

Listing 5.3: Code for smoothing positive and negative labels. We used two methods, one for positive labels and one for negative labels for simplicity, but obviously these operations can be generalized into a single method.

Training

The network was trained from scratch. Weights were initialized from a Gaussian distribution with mean 0 and standard deviation 0.02. We set the number of epochs to 400, the batch size to 12 and the learning rate to $2 * 10^{-4}$. Adam optimizer was used with $\beta_1 = 0.5$ and $\beta_2 = 0.999$. The `L1_LAMBDA` for the L1 loss of the generator was set to `L1_LAMBDA` = 100. The training for about 1000 images was done using a Tesla P100-PCIE-16GB and it took about 8 hours. Figures 5.4 and 5.5 show the trends of generator and discriminator losses during the training. Results were really good and promising: they are shown in chapter 6.

5.2.2 Training on UniToPatho

Training the GAN on PanNuke was not particularly difficult and good quality results were obtained without having to do a challenging hyperparameter search. Instead, the same cannot be said for UniToPatho, where the task has been more difficult for a number of reasons which we will describe below.

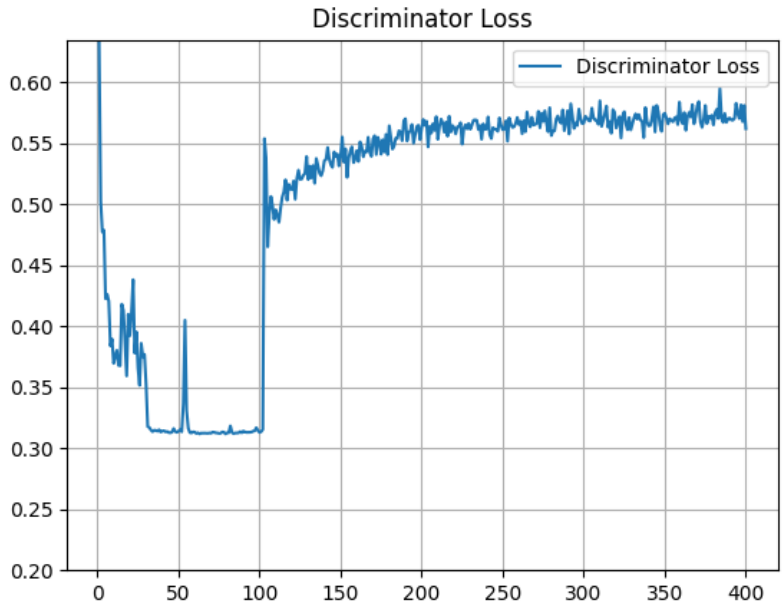


Figure 5.4: The trend of the discriminator loss during the training on PanNuke.

.

Images resolution

First of all, UniToPatho is a much larger dataset. On one hand we have PanNuke, from which we had about 1100 images of the colon tissue with a resolution of 256x256 pixels. On the other hand we have UniToPatho, containing 7174 images (5103 in the training set and 2071 in the test set) with a resolution of 1812x1812 pixels. Figure 5.6 shows some examples of images of UniToPatho.

The first problem to be faced was that the images to be processed do not have a resolution equal to a power of 2. In fact, a Pix2pix GAN uses a U-Net for the generator and a U-Net typically downsamples by a factor of 2 in the encoder and upsamples by a factor of 2 in the decoder. Therefore, working with images which do not have a resolution of a power of 2 can lead to artifacts during the downsampling or upsampling process. To solve this problem, several strategies have been devised:

- Resize the image to the power of 2 closest to 1812, i.e. 2048.
- Set a padding to bring the image to 2048x2048.

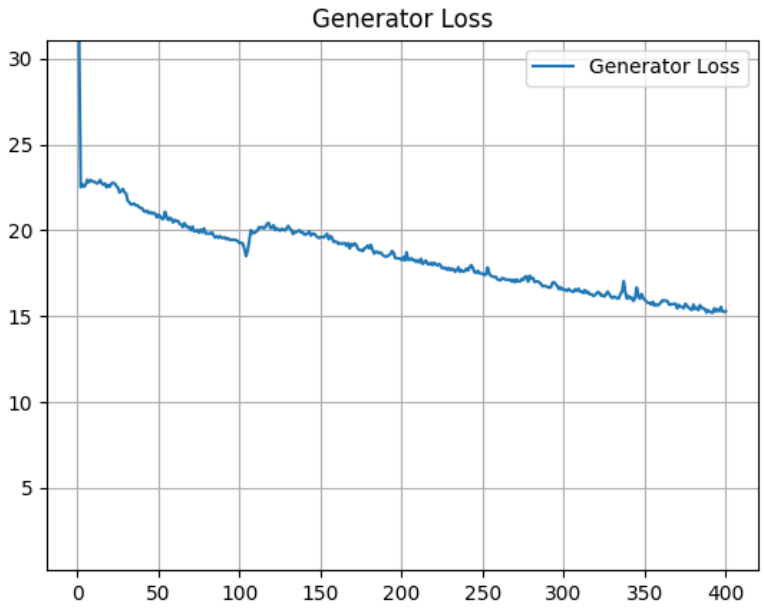


Figure 5.5: The trend of the generator loss during the training on PanNuke.

- Crop the image with a size of a power of 2, for example 256, 512 or 1024.

The resize was discarded as it was thought that it could compromise the features of the cells. In fact, Torchvision provides several interpolation techniques to resize the images and studying how the different techniques could affect the cell features seemed too complicated to us. Besides, training with 2048x2048 images inevitably takes too much time. Resize to a smaller size (e.g. 1024 or 512) was also considered, but this option was also discarded as it would still have resulted in information loss and cells size distortion.

Similarly, the padding option was discarded. In fact, we could have opted for a padding with a constant value (for example, 0, i.e. black) which would have formed a sort of frame in the images, but this was not good since the GAN would then also have learned to draw the frame. Or we could have used a *reflect* padding, but with this method the shape of the cells arranged on the border of the images are changed and, consequently, we were afraid that even the GAN would have learned to draw cells with distorted shape.

At the end, the choice fell on the third option, that is to use the crop transformation. Tests were made with center crop and with random crop, but the best results were obtained using the Torchvision *FiveCrop* transformation, which re-

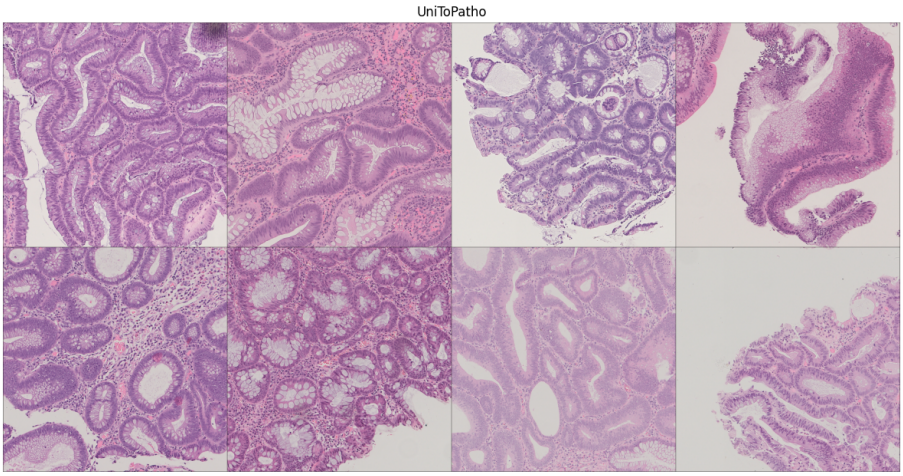


Figure 5.6: Some examples of images of UniToPatho used to train the GAN.

turns, for each image, five crops: four corresponding to the four corners plus one in the center. The size chosen for the crops was 256x256 pixels, which allowed us to obtain quite satisfactory performances in terms of time spent for the trainings.

Manage the dataset

Another very annoying problem was that the images, as they have a high resolution, they also have a large size on disk, and this slows down the whole training process. In fact, the images are saved as `.png` files and usually have a size of about 6-7 MB, while the masks are also saved in `.png` but they have a much smaller size, in the order of a few hundreds of KB (the `png` encoding lends itself very well to segmentation masks, which typically present a few different colors). Consider that for each training sample, the following operations must be performed:

1. Read the `.png` image file from disk to obtain a tensor of shape `(H, W, 3)`.
2. Read the `.png` mask file from disk to obtain a tensor of shape `(H, W, 3)`.
3. Transform the mask from a tensor of shape `(H, W, 3)` into a tensor of shape `(H, W, classes)`, where `classes` is the number of classes and each pixel is labeled with a one-hot vector.

4. Perform the transformations related to the data augmentation. The transformations were: FiveCrop, Flip (both horizontal and vertical) and RandomRotate90 (it rotates the image 90 degrees 0 or more times).

Initially, performing these operations without any tricks took almost 1 second for each training example: far too much for a training set of more than 5000 images. Performance on operations 1 and 2 are difficult to improve, as they consist of a reading from the disk and the performances depend fundamentally on the hardware used. `CV2` and `PIL` were used for reading the files, which typically have good performances.

Instead, for operations 3 and 4 an unusual decision was taken but which brought about clear improvements: performing the transformations directly on CUDA tensors (i.e. tensors working on GPU). Typically, in fact, when a Python class for a dataset is built in PyTorch, one works with CPU tensors (or sometimes even NumPy arrays, for example we used NumPy arrays to do data augmentation with Albumentation in the segmentation task). However, this made the process too onerous, particularly in operation 3. The use of CUDA tensors significantly improved the performances. The disadvantage of this choice was that we had to give up Albumentations, which works on NumPy arrays (which makes the library particularly flexible and usable with most ML frameworks).

In order to do data augmentation we thus used the Torchvision package and the main difficulty was to ensure that the transformations were applied to the image and the mask in the same way. The solution to solve this problem was quite simple for the transformations that we used: concatenate the image and the mask along the channel dimension to obtain a single tensor and perform the transformations on it. To clarify this last aspect, we provide a sketch of code that performs a random crop on image and mask in the same way:

```
import torch
from torchvision.transforms import RandomCrop

img = < image with shape (C, H, W) >
mask = < mask with shape (CLASSES, H, W) >

# Concat img and mask along the channel dimension.
stacked_image = torch.cat([img, mask], dim=0) # (C + CLASSES, H, W)

# Crop at a random location.
transformed = RandomCrop(256)(stacked_image)
cropped_img, cropped_mask = transformed[:C, ...], transformed[C:, ...]
```

Listing 5.4: Sketch of how to do random crop on both image and mask concatenating them along the channel dimension. Clearly, other transformations can be applied with this method.

So, the transformation Flip() of Albumentations was replaced by RandomHorizontalFlip() and RandomVerticalFlip() of Torchvision. Instead, the transformation RandomRotate90() is not provided in Torchvision, so we defined a simple implementation of it, which we report below:

```
class RandomRotate90(object):
    """Randomly rotate the input by 90 degrees zero or more times."""
    def __init__(self, p=0.5):
        self.p = p

    def __call__(self, img):
        if random.random() < self.p:
            return torch.rot90(img, random.randint(0, 3), dims=(1, 2))
        return img
```

Listing 5.5: Our implementation of RandomRotate90.

One could also have thought of saving the training set directly in NumPy arrays with file extension `.npy` instead of `.png`. Actually, doing this makes the process even slower since a `.npy` file of an 1812x1812x6 mask weighs about 70-80 MB, too much to be read in a reasonable time.
Another solution could have been to read all the files before the training and keep all the tensors in RAM. This choice was discarded as it would have required several hundreds GB of RAM, which we did not have.

Multi-GPU training

Another very useful trick that allowed us to further increase performances was to use multiple GPUs in parallel to train the GAN. PyTorch provides a module called *DistributedDataParallel* (*DDP*) which implements data parallelism. Multiprocessing with DistributedDataParallel duplicates the model across multiple GPUs, each of which is usually controlled by one process. During training, each process loads its own minibatches from disk and passes them to its GPU. Each GPU does its own forward pass, and then the gradients are all-reduced across the GPUs. Gradients for each layer do not depend on previous layers, so the gradient all-reduce is calculated concurrently with the backwards pass to further alleviate the networking bottleneck. At the end of the backwards pass, every node has the averaged gradients, ensuring that the model weights stay synchronized. In addition, each process needs to know which slice of the data to work on so that the batches are non-overlapping: Pytorch provides *DistributedSampler* to accomplish this.

Ground truth segmentation masks are not available

Another problem that was difficult to face is the fact that we did not have the ground truth segmentation masks for UniToPatho, while we did on PanNuke. The masks with which the GAN was trained are masks obtained by making inference with a neural network trained on PanNuke, so it is difficult, without having the ground truth masks, to evaluate if the masks produced by the neural network are actually precise and qualitative. A trick that led to clear improvements in producing the masks was to note that the coloring of the UniToPatho images was quite different from those of PanNuke. Making inference on images that have a different color, in fact, means using the model on a distribution of data rather different from that in which the network was trained and therefore the results can be poorer than the expected ones.

The reason of this different coloring lies in the fact that in many biological fields, tissue samples are taken from a subject for analysis and one common way of analyzing the tissue sample is to treat it with stains that have selective affinities for different biological substances. The majority of stains only absorb light, and the stained slides are therefore viewed using a microscope with a light illuminating the sample from below. If no stain is present, all of the light will pass through, appearing bright white. Areas where the stain has adhered to a substance in the tissue will absorb some of the light. The amount of light absorbed depends on many factors. Moreover the absolute color values of a slide have many influences, only one of which is the biological component. This biological component is the actual amount of the cellular substance to which a particular stain will attach. For example, in the most popular staining method for medical diagnosis, *hematoxylin* selectively stains nucleic acids a blue-purple hue while *eosin* stains proteins a bright pink color [47].

The solution was to use the histological slide normalization method described in [47], whose implementation on PyTorch is called Torchstain and was retrieved from [15]. This led to a great improvement in the production of the masks: figure 5.7 shows the difference between an original UniToPatho image and the same normalized one, with the relative segmentation masks obtained.

Trials and errors

Let's now get more into the aspects strictly related to the training. In the first training attempts, a setting almost identical to that used for PanNuke was used, with the hope of obtaining similar results. Despite the similar setting, the images produced by the GAN at the end of the training were decidedly worse than those obtained on PanNuke. A trivial mistake but rather difficult to debug was that the `BatchNorm` layer, when using multi-GPU training, must be replaced by the `SyncBatchNorm` layer. Below we put the code we used to

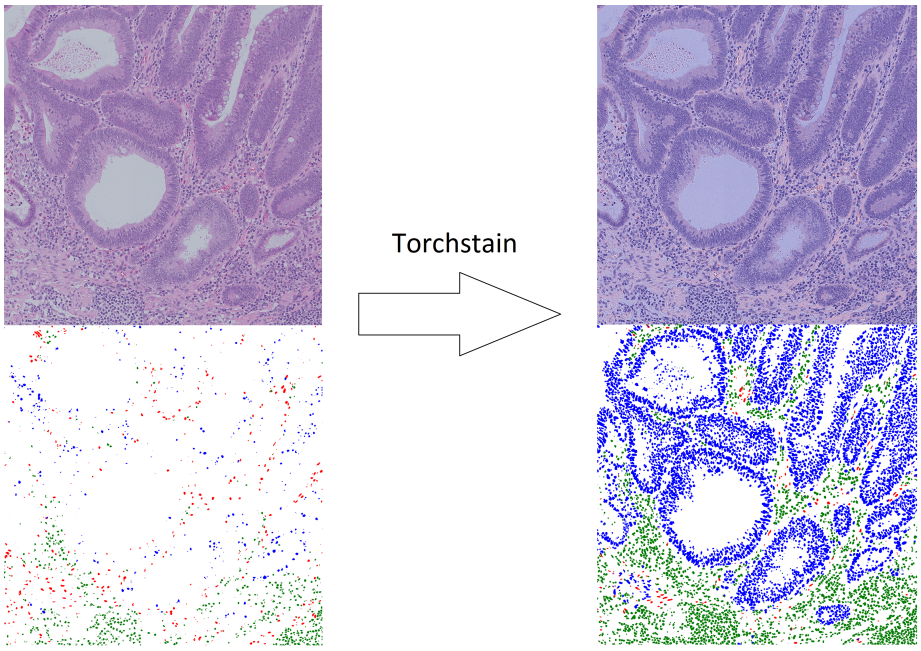


Figure 5.7: On the left an original image of UniToPatho with the obtained segmentation mask. On the right the same image normalized with Torchstain, with the obtained segmentation mask. As you can see, cell segmentation is much better thanks to the normalization.

convert `BatchNorm` layers to `SyncBatchNorm` in our models.

```
disc = Discriminator(...)
gen = Generator(...)
# Use SynchBatchNorm for Multi-GPU trainings
disc = torch.nn.SyncBatchNorm.convert_sync_batchnorm(disc)
gen = torch.nn.SyncBatchNorm.convert_sync_batchnorm(gen)
```

After solving this initial problem, subsequent trainings showed better results but soon another problem arose. In fact, in the training script, at the beginning 10 examples are extracted from the test set (therefore examples never seen by the GAN) and after each epoch the images generated by the GAN for these 10 examples are logged, in order to show the trend of the images produced during the training. In the early epochs, the training seemed to proceed in the right direction, in fact after each epoch the network improved and the quality of the images increased. However, from a certain point of training onwards, the GAN began to produce visibly synthetic images, very distant from the real ones, because it produced a strange effect, that added a particular coloring in the images, which we would define "purple fog". Figure 5.8 shows an example

of this strange coloring that GAN produced.

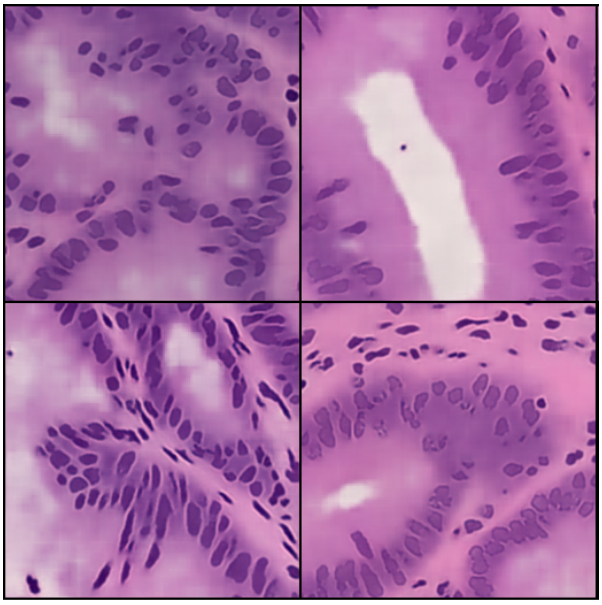
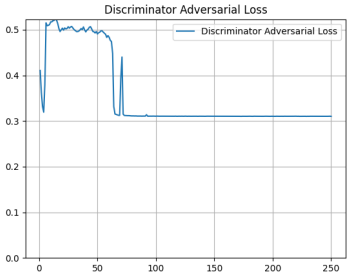


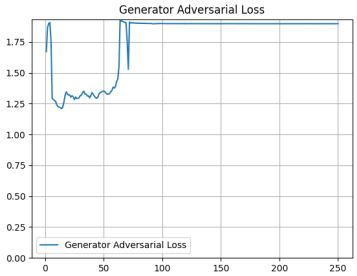
Figure 5.8: GAN produced after some epochs images affected by the "purple fog" effect.

The resolution of this problem was found by carefully examining the trend of the losses. Figure 5.9 shows the trend of the losses of the generator (both L1 and adversarial) and of the discriminator. As you can easily see, at a certain point (about at the epoch 60), the L1 loss undergoes a sharp decrease, which at the same time leads to a decrease in the discriminator loss and a corresponding increase in the adversarial loss in the generator. Observing the images that were logged from epoch to epoch, it was noticed that from that moment the generator began to insert that strange effect of "purple fog" in the images: clearly this coloring means that the generator is no longer generating realistic images and the discriminator recognizes fake images very easily.

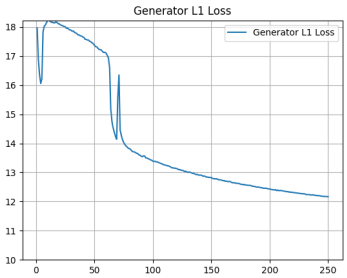
Then, it was clear that the problem was due to the L1 loss and that it was necessary to adjust the `L1_LAMBDA` parameter, which regulates the L1 loss of the generator. Remember the formulation of the generator loss expressed in 5.6. Initially set to `L1_LAMBDA = 100`, with which excellent results have been obtained on PanNuke, clear improvements on UniToPatho have been obtained by reducing it. Tests were carried out with different values, and the best setting was obtained with the value `L1_LAMBDA = 75`. In this way, the "purple fog" disappeared from the generated images.



(a) Trend of the discriminator adversarial loss.

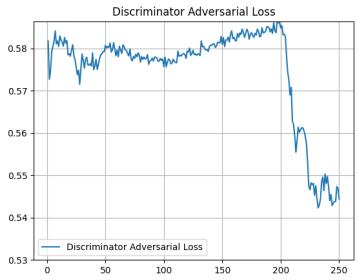


(b) Trend of the generator adversarial loss.

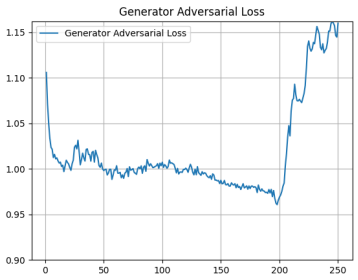


(c) Trend of the generator L1 loss.

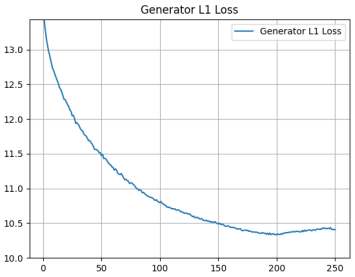
Figure 5.9: The trends of the losses of the discriminator and of the generator when the "purple fog" was produced. On the x-axis there are the epochs, on the y-axis there are the loss values. When the L1 loss goes down (more or less at the epoch 60) the GAN starts to produce the "purple fog" effect.



(a) Trend of the discriminator adversarial loss.



(b) Trend of the generator adversarial loss.



(c) Trend of the generator L1 loss.

Figure 5.10: The trends of the losses in the best training.

Training

Trainings were performed with the pretrained model on PanNuke and with the model trained from scratch, but the best results were obtained with the pretrained model. The best run was obtained with the following setting.
The network was trained from the pretrained model on PanNuke. We set the number of epochs to 250 and the learning rate to $2{*}10^{-4}$. We used the FiveCrop transformation, so we took from each image five corresponding crop of size 256 and consequently the actual size of the training set must be multiplied by 5: $5103 * 5 = 25515$. Similarly, the batch size was set to 2 but the effective batch size was $2 * 5 = 10$. Adam optimizer was used with $\beta_1 = 0.5$ and $\beta_2 = 0.999$. The `L1_LAMBDA` for the L1 loss of the generator was set to `L1_LAMBDA` $= 75$. The training was done using two Tesla T4-16GB in parallel and it took about 1 day and 16 hours. Figure 5.10 show the trends of the generator and discriminator losses during the training. Results are shown in chapter 6.

Chapter 6

Results

In this chapter we will talk about the results we achieved in our work.

6.1 Segmentation

Since this work focuses only about the generation task, we will show below only some examples of semantic segmentation masks produced by our model, but in my colleague Davide Di Luccio's work [45] you can find more examples and also some interesting data about metrics scores obtained.

Figures 6.1 and 6.2 show some examples of segmentation mask obtained on PanNuke: on the left there is the image, in the center the predicted segmentation mask and on the right the ground truth segmentation mask.

Figure 6.1: Example 1 of segmentation on PanNuke.

Figure 6.2: Example 2 of segmentation on PanNuke.

Instead, figures 6.3, 6.4 and 6.5 show some examples of semantic segmentation masks obtained on UniToPatho dataset.

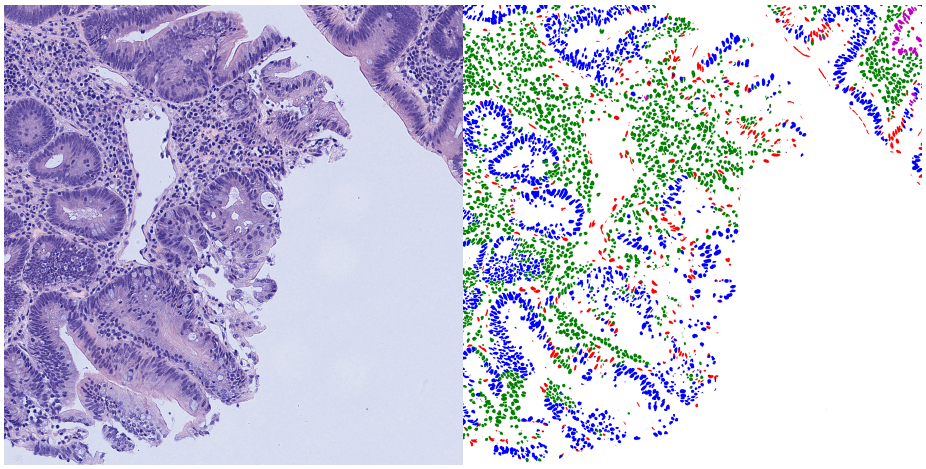


Figure 6.3: Example 1 of segmentation on UniToPatho.

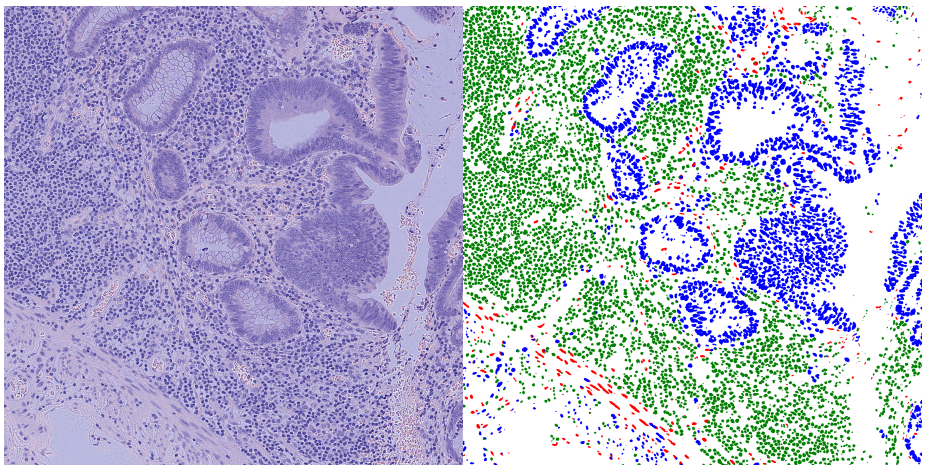


Figure 6.4: Example 2 of segmentation on UniToPatho.

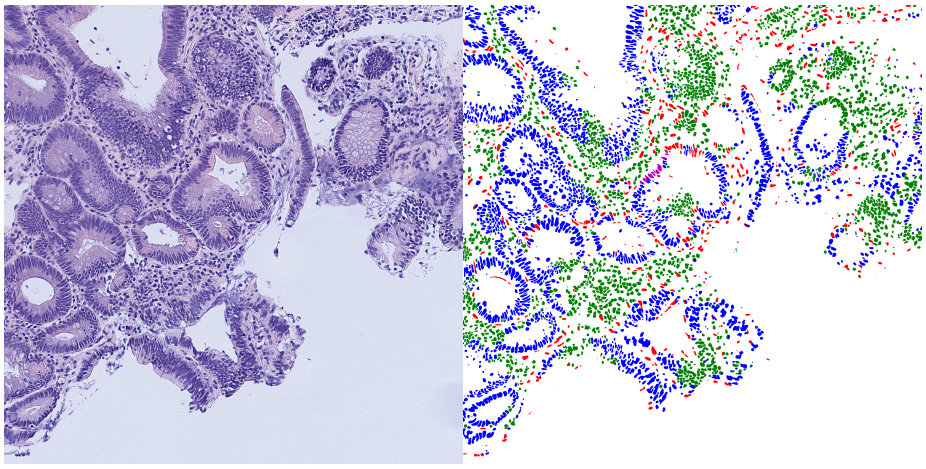


Figure 6.5: Example 3 of segmentation on UniToPatho.

6.2 Generation

6.2.1 Results on PanNuke

First of all we show some results obtained with the GAN trained on PanNuke with the setting defined in the section 5.2.1 at the paragraph "Training". In the figure 6.6 we show some examples of synthetic images produced, where each image has size 256x256 pixels.

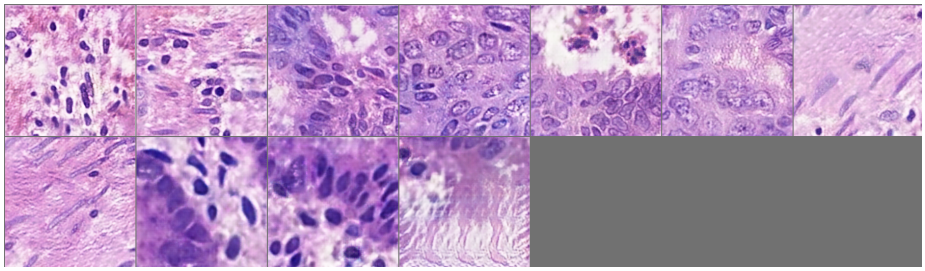


Figure 6.6: Some synthetic images produced by the GAN trained on PanNuke.

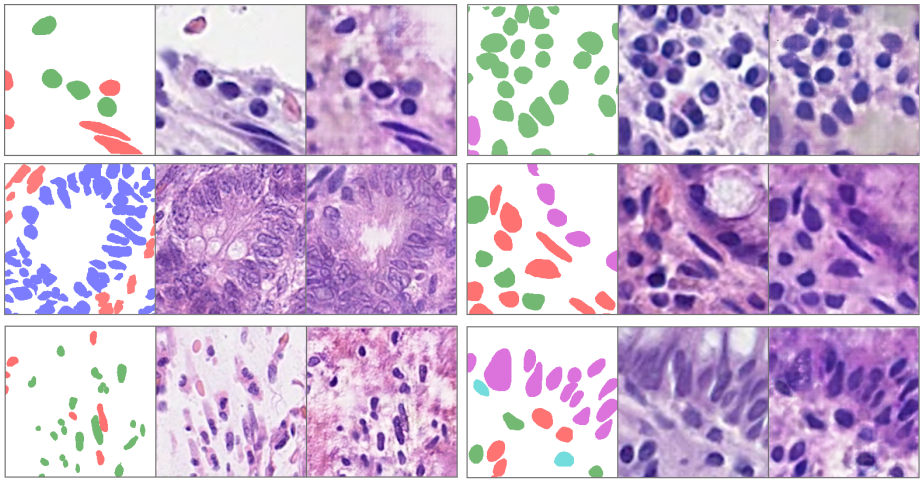


Figure 6.7: Some examples where you can compare the real image and the synthetic image produced by the GAN starting from the segmentation mask. On the left there is the segmentation mask and beside the real image and the fake image. Have you tried to guess if the fake images are in the center or on the right? We hope that your choice was difficult to take. In the center there are the real images and on the right the fake images.

As you can see the images are quite realistic. A good way to evaluate the quality of the images produced is to compare the synthetic images with the real ones.

Figure 6.7 shows some examples in which on the left there is the segmentation mask, and then beside to it, there are the real image and the synthetic image (not necessarily in this order). The caption of the image says where the synthetic images are positioned (right or center). Before reading it, try to look at them and guess which are the fake ones and the real ones.

6.2.2 Results on UniToPatho

After training the GAN on PanNuke, the first experiment we did was to try to make inference with the GAN trained on PanNuke on the UniToPatho segmentation masks, since the datasets are different but still similar. The figure 6.8 shows the obtained results: the produced images are quite distant from those of UniToPatho, but we have to take into account that the GAN has never seen images of UniToPatho. From this observation we understood that we had to train the GAN on UniToPatho as well.

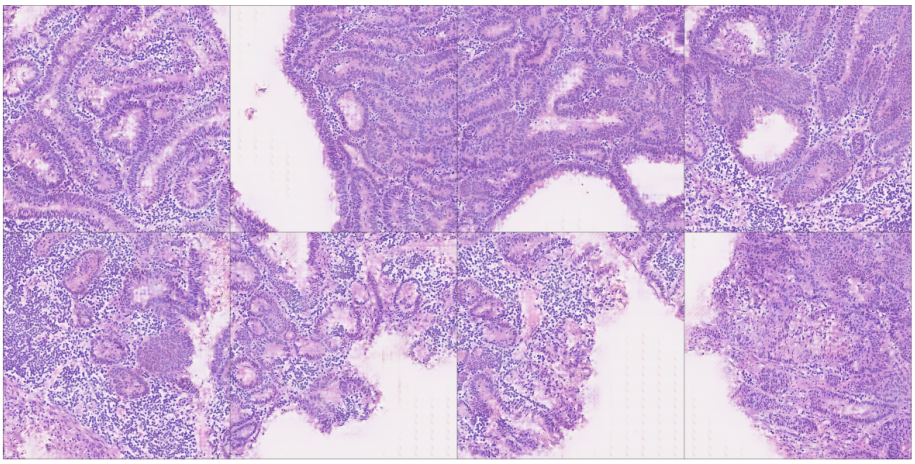


Figure 6.8: Some results obtained with the GAN trained on PanNuke. We took some images from the test set of UniToPatho and we ran the GAN in inference mode.

Well, now we will show the results we obtained with the GAN trained on UniToPatho. As described in section 5.2.2, it was quite difficult to train the GAN and the results we will show in the next figures come from the GAN trained with the setting described in section 5.2.2 in the paragraph "Training". In the figure 6.9 we show some synthetic images produced by our GAN: each image in the figure has size 1812x1812 pixels. To produce these images, we started from the segmentation masks of some images belonging to the test set of UniToPatho. These masks have a resolution of 1812x1812 pixels. Remember that the GAN has a U-Net for a generator and that a U-Net produces artifacts during the

downsampling/upsampling process if it does not work with images that have size of a power of 2. So, we added a *reflect* padding to bring the masks to 2048x2048. Then we generated the synthetic images with the GAN and, finally, in order to bring the images back to the original resolution of 1812x1812, we use a CenterCrop of size 1812.

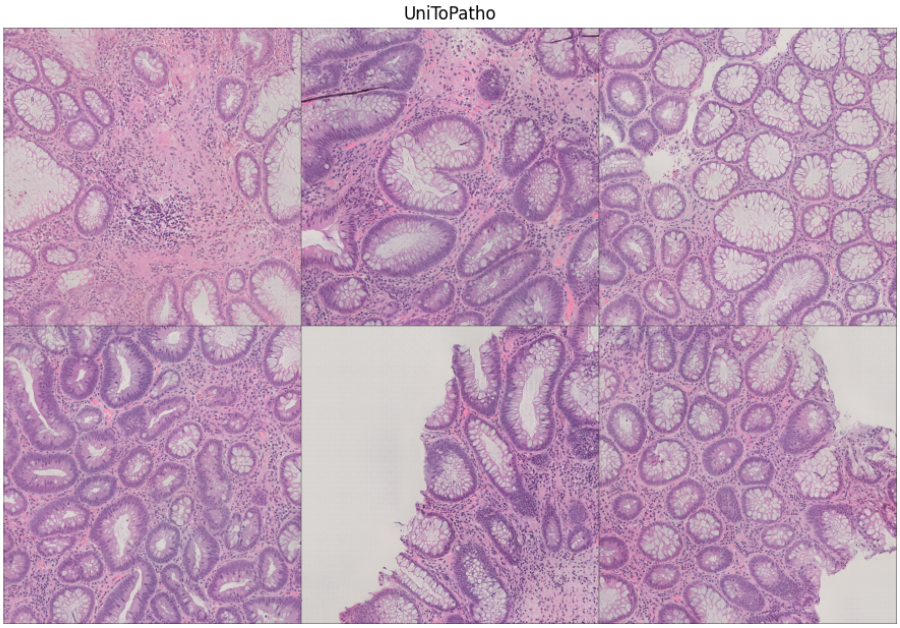


Figure 6.9: Some synthetic images produced by the GAN trained on UniToPatho.

As for the results on PanNuke, we put some figures (from figure 6.10 to figure 6.17) in which there is the segmentation mask on the left, the real image in the center and the synthetic image on the right. This test samples were taken with a RandomCrop of size 1024px from some images coming from the test set of UniToPatho.

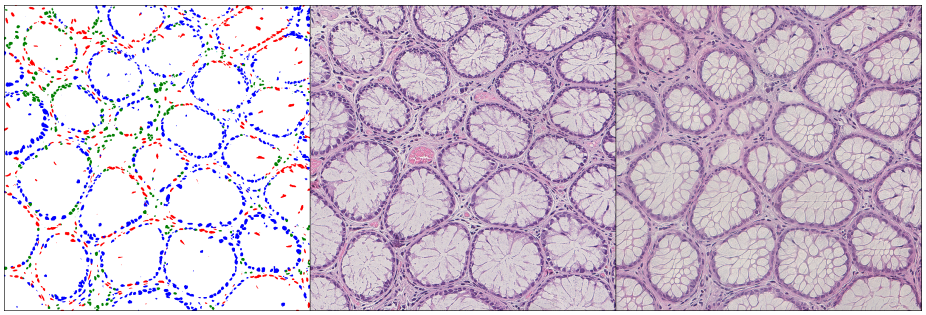


Figure 6.10: Example 1 of comparison between real and synthetic image.

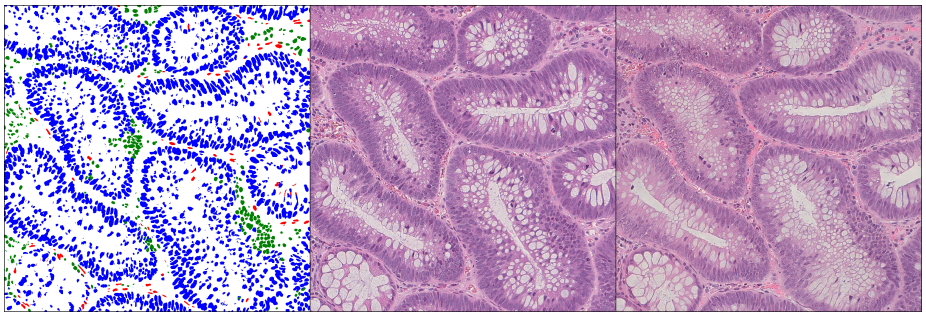


Figure 6.11: Example 2 of comparison between real and synthetic image.

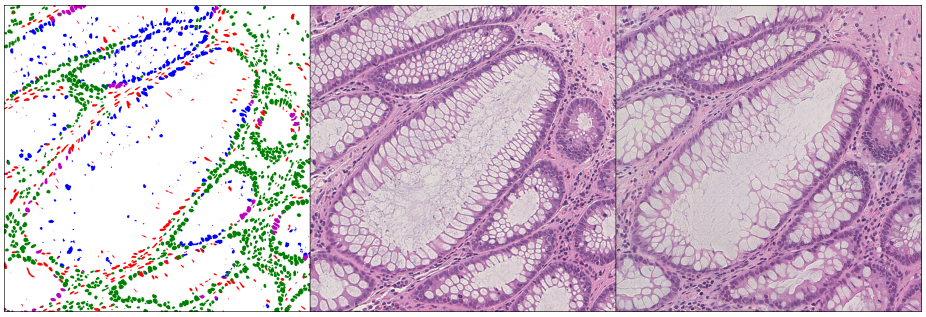


Figure 6.12: Example 3 of comparison between real and synthetic image.

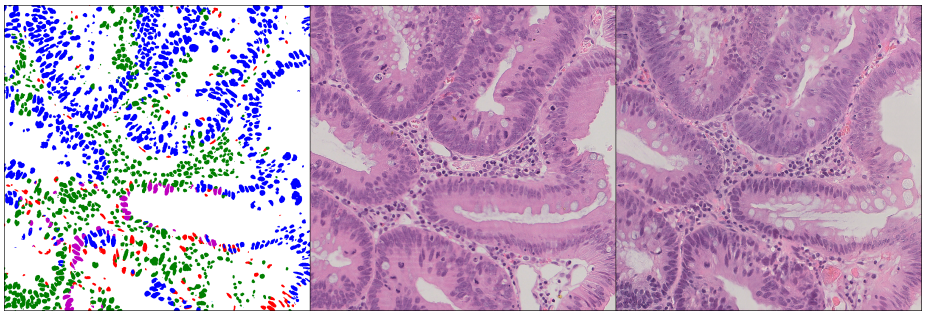


Figure 6.13: Example 4 of comparison between real and synthetic image.

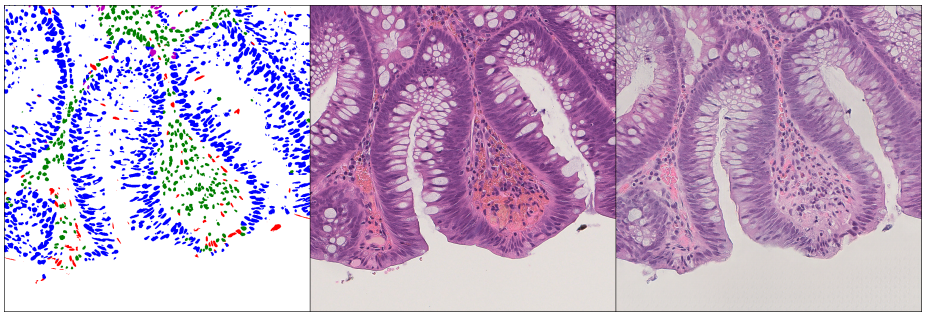


Figure 6.14: Example 5 of comparison between real and synthetic image.

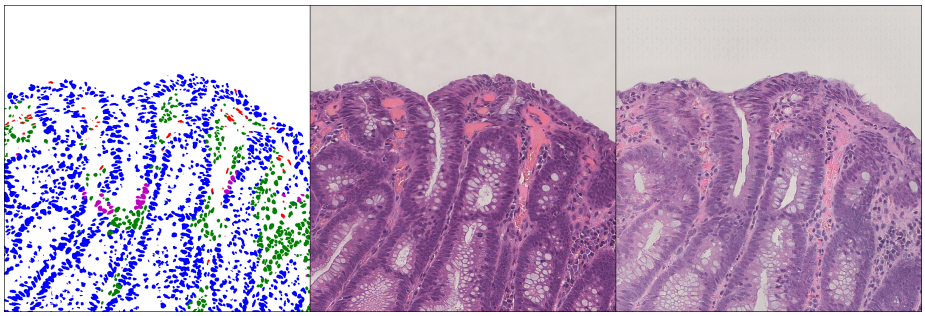


Figure 6.15: Example 6 of comparison between real and synthetic image.

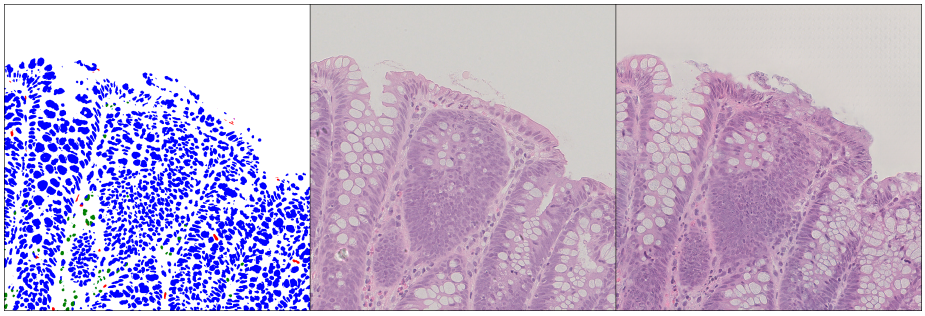


Figure 6.16: Example 7 of comparison between real and synthetic image.

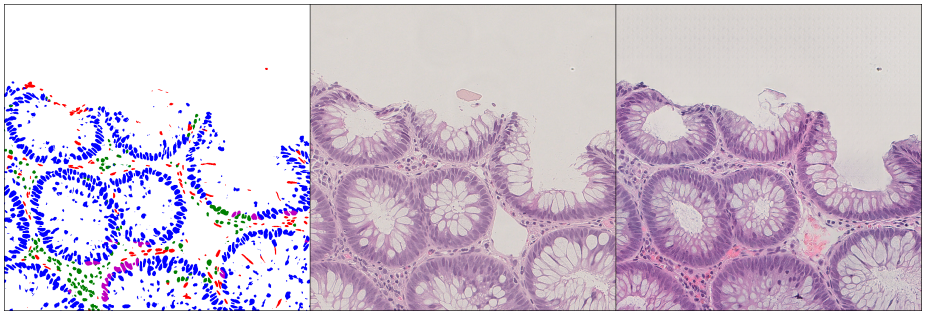


Figure 6.17: Example 8 of comparison between real and synthetic image.

Chapter 7

Conclusions and future work

In this work we studied the problem of how to do data augmentation on the medical dataset UniToPatho. Since doing data augmentation on a medical dataset such as UniToPatho can produce annoying artifacts with traditional techniques, we explored the solution of doing data augmentation with an alternative approach, i.e. to generate new data with a generative model.

So, in this work we first developed a U-Net on PanNuke, with the aim of producing the segmentation masks of UniToPatho. Once produced these masks, we used them for developing a Pix2pix GAN, able to produce new histopathological samples starting from the segmentation masks.

At this point, we tried to do some tests to verify the realism of the synthetic images.

At first, we produced for each image belonging to the test set of UniToPatho its corresponding synthetic. Then, we took the classification models built from the original dataset and we calculated the accuracy of these models for the *real* test set and for the *synthetic* test set. The accuracy of the real test set was about 81%, while for the synthetic test set the models achieved an accuracy of about 73%. As it was easy to expect, we had a reasonable drop of the accuracy, but we can say that the result has been quite satisfying.

Then, we asked a pathologist to annotate some real or fake images. In particular, we used an annotation tool that showed the pathologist 40 random images (20 fake and 20 real), for which he was asked to indicate:

- Real or fake
- High-grade or low-grade dysplasia

- Tubular or tubulo-villous adenoma

Once concluded the annotations, we calculated the following results. The 60% (12/20) of the synthetic images were labelled as real. The accuracy for the real/fake labels was 60%. Regarding the distinction between low-grade and high-grade dysplasia, the accuracy on only the real images was 70%, while on the fake images was 80%. So, we can say that probably the synthetic images preserve the diagnostic information for detecting the low or high grade of dysplasia. Instead, for the the distinction between tubular and tubulo-villous adenoma, the pathologist told we that this information is very difficult to take just by looking the image (more information is needed), so we did not give too much importance to it.

Regarding the future works, the most interesting thing to do would certainly be to generate new synthetic samples thought to be included in UniToPatho. We said several times that if we generate synthetic images from segmentation masks we can produce highly precise samples. Well, in this way it would be interesting to produce new synthetic samples and label them, and then add them to the original dataset. By doing so, with a richer dataset we expect Deep Learning algorithms to be more likely to perform better.

Bibliography

[1] URL: https://en.wikipedia.org/wiki/Colorectal_polyp..

[2] URL: https://en.wikipedia.org/wiki/Dysplasia.

[3] URL: https://becominghuman.ai/multi-layer-perceptron-mlp-models-on-real-world-banking-data-f6dd3d7e998f.

[4] URL: https://ai.stackexchange.com/questions/11567/what-causes-a-model-to-require-a-low-learning-rate.

[5] URL: https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60.

[6] URL: https://github.com/cazala/mnist.

[7] URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[8] URL: https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/.

[9] URL: https://developers.google.com/machine-learning/gan/generative.

[10] URL: https://www.jeremyjordan.me/semantic-segmentation/.

[11] URL: https://nanonets.com/blog/semantic-image-segmentation-2020/.

[12] URL: https://albumentations.ai/.

[13] URL: https://docs.wandb.ai/guides/sweeps.

[14] URL: https://pytorch.org/docs/stable/generated/torch.nn.Unfold.html.

[15] URL: https://github.com/EIDOSlab/torchstain.

[16] URL: https://pytorch.org/vision/stable/index.html.

[17] URL: https://numpy.org/.

[18] URL: https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html.

[19] Alexander Amini et al. “Spatial Uncertainty Sampling for End-to-End Control”. In: (May 2018).

[20] Carlo Alberto Barbano et al. “UniToPatho, a labeled histopathological dataset for colorectal polyps classification and adenoma dysplasia grading”. In: *arXiv preprint arXiv:2101.09991* (2021).

[21] Thomas Bel et al. “Residual CycleGAN for robust domain transformation of histopathological tissue slides”. In: *Medical Image Analysis* 70 (Feb. 2021), p. 102004. DOI: 10.1016/j.media.2021.102004.

[22] A. Ben Hamida et al. “Deep learning for colon cancer histopathological images analysis”. In: *Computers in Biology and Medicine* 136 (2021), p. 104730. ISSN: 0010-4825. DOI: https://doi.org/10.1016/j.compbiomed.2021.104730. URL: https://www.sciencedirect.com/science/article/pii/S0010482521005242.

[23] Roisin Bevan and Matthew Rutter. “Colorectal Cancer Screening—Who, How, and When?” In: *Clinical Endoscopy* 51 (Jan. 2018), pp. 37–49. DOI: 10.5946/ce.2017.141.

[24] Mehdi Cherti. “Deep generative neural networks for novelty generation : a foundational framework, metrics and experiments”. PhD thesis. Jan. 2018.

[25] Richard Colling et al. “Artificial intelligence in digital pathology: A roadmap to routine use in clinical practice”. In: *The Journal of Pathology* 249 (May 2019). DOI: 10.1002/path.5310.

[26] Corinna Cortes and Vladimir Vapnik. “Support-vector networks”. In: *Machine learning* 20.3 (1995), pp. 273–297.

[27] Ramzi S. Cotran, Vinay Kumar, and Stanley L. Robbins. “Chap. 17”. In: *Robbins and COTRAN PATHOLOGIC basis of disease*. Elsevier, Saunders, 2005.

[28] Johannes Fürnkranz. “Decision Tree”. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 263–267. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_204. URL: https://doi.org/10.1007/978-0-387-30164-8_204.

[29] Jevgenij Gamper et al. “PanNuke Dataset Extension, Insights and Baselines”. In: (Mar. 2020).

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

[31] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[32] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 1998, p. 842. ISBN: 978-0-13-273350-2.

[33] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: 7 (Dec. 2015).

[34] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: 7 (Dec. 2015).

[35] Geoffrey Hinton. *Geoffrey Hinton Neural Networks for machine learning nline course.* URL: https://www.coursera.org/learn/neural-networks/home/welcome.

[36] “Histopathology classification and localization of colorectal cancer using global labels by weakly supervised deep learning”. In: *Computerized Medical Imaging and Graphics* 88 (2021), p. 101861. ISSN: 0895-6111. DOI: https://doi.org/10.1016/j.compmedimag.2021.101861. URL: https://www.sciencedirect.com/science/article/pii/S0895611121000094.

[37] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* 2015. arXiv: 1502.03167 [cs.LG].

[38] Phillip Isola et al. *Image-to-Image Translation with Conditional Adversarial Networks.* 2018. arXiv: 1611.07004 [cs.CV].

[39] Mostafa Jahanifar, Navid Koohbanani, and Nasir Rajpoot. “NuClick: From Clicks in the Nuclei to Nuclear Boundaries”. In: (Sept. 2019).

[40] Tero Karras, Samuli Laine, and Timo Aila. *A Style-Based Generator Architecture for Generative Adversarial Networks.* 2019. arXiv: 1812.04948 [cs.NE].

[41] Jakob Kather et al. “Multi-class texture analysis in colorectal cancer histology”. In: *Scientific Reports* 6 (June 2016), p. 27988. DOI: 10.1038/srep27988.

[42] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2017. arXiv: 1412.6980 [cs.LG].

[43] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[44] Yann Lecun, Patrick Haffner, and Y. Bengio. “Object Recognition with Gradient-Based Learning”. In: (Aug. 2000).

[45] Davide Di Luccio. “Semantic segmentation of histopathological tissue with Deep Learning”. In: (Oct. 2021).

[46] Agnes Lydia and Sagayaraj Francis. “Adagrad - An Optimizer for Stochastic Gradient Descent”. In: Volume 6 (May 2019), pp. 566–568.

[47] Marc Macenko et al. “A Method for Normalizing Histology Slides for Quantitative Analysis.” In: vol. 9. June 2009, pp. 1107–1110. DOI: 10.1109/ISBI.2009.5193250.

[48] Mehdi Mirza and Simon Osindero. *Conditional Generative Adversarial Nets.* 2014. arXiv: 1411.1784 [cs.LG].

[49] Kunal Nagpal et al. “Development and validation of a deep learning algorithm for improving Gleason scoring of prostate cancer”. In: *npj Digital Medicine* 2 (June 2019), p. 48. DOI: 10.1038/s41746-019-0112-2.

[50] A. B. Novikoff. "On Convergence Proofs on Perceptrons". In: *Proceedings of the Symposium on the Mathematical Theory of Automata*. Vol. 12. Polytechnic Institute of Brooklyn, 1962, pp. 615–622.

[51] Luke Oakden-Rayner et al. "Hidden Stratification Causes Clinically Meaningful Failures in Machine Learning for Medical Imaging". In: (Sept. 2019).

[52] Alec Radford, Luke Metz, and Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2016. arXiv: 1511.06434 [cs.LG].

[53] Saima Rathore et al. "Segmentation and Grade Prediction of Colon Cancer Digital Pathology Images Across Multiple Institutions". In: *Cancers* 11 (Nov. 2019), p. 1700. DOI: 10.3390/cancers11111700.

[54] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *LNCS* 9351 (Oct. 2015), pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28.

[55] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain". In: *Psychological Review* (1958), pp. 65–386.

[56] Muhammad Shaban et al. "Prognostic significance of automated score of tumor infiltrating lymphocytes in oral cancer." In: *Journal of Clinical Oncology* 36 (May 2018), e18036–e18036. DOI: 10.1200/JCO.2018.36.15_suppl.e18036.

[57] Korsuk Sirinukunwattana et al. "Locality Sensitive Deep Learning for Detection and Classification of Nuclei in Routine Colon Cancer Histology Images". In: *IEEE Transactions on Medical Imaging* 35 (Feb. 2016), pp. 1–1. DOI: 10.1109/TMI.2016.2525803.

[58] Carole Sudre et al. "Generalised Dice overlap as a deep learning loss function for highly unbalanced segmentations". In: (July 2017).

[59] Min-Jen Tsai and Yu-Han Tao. "Deep Learning Techniques for the Classification of Colorectal Cancer Tissue". In: *Electronics* 10.14 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10141662. URL: https://www.mdpi.com/2079-9292/10/14/1662.

[60] Jason Tseung. "Robbins and Cotran Pathologic Basis of Disease: 7th Edition". In: *Pathology* 37 (Apr. 2005), p. 190. DOI: 10.1080/00313020500059191.

[61] Walter Pitts Warren S McCulloch. *"A logical calculus of the ideas immanent in nervous activity*. 1943.

[62] Pavel Yakubovskiy. *Segmentation Models Pytorch*. https://github.com/qubvel/segmentation_models.pytorch. 2020.

[63] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012. arXiv: 1212.5701 [cs.LG].

[64] Zongwei Zhou et al. "UNet++: A Nested U-Net Architecture for Medical Image Segmentation". In: (July 2018).