Escola Tècnica Superior
d'Enginyeria de Telecomunicació de
Barcelona

Data-driven Confocal Microscopy to Hematoxylin and Eosin Transformation

by

Sergio García Campderrich

In partial fulfilment of the requirements for the degree in Telecommunications Technologies and Services Engineering

supervised by

Verónica Vilaplana

October 6, 2019

Abstract

CM... DL...

Acknowledgments

Thanks to...

Revision history and approval record

Revision	Date	Purpose
0	September 2, 2019	Document creation
1	October 1, 2019	Document revision
2	October 5, 2019	Document modification
3	October 6, 2019	Document revision

Name	E-mail
Sergio García Campderrich	sergio.garcia.campderrich@estudiant.upc.edu
Verónica Vilaplana Besler	veronica.vilaplana@upc.edu

	Written by		Reviewed and approved by
Date	October 5, 2019	Date	October 6, 2019
Name	Sergio García Campderrich	Name	Verónica Vilaplana Besler
Position	Project author	Position	Project supervisior

Contents

List of Figures

List of Tables

Acronyms

ANNs Artificial Neural Networks.

CM Confocal Microscopy.

CNNs Convolutional Neural Networks.

CycleGANs Cycle-Consistent Generative Adversarial Networks.

DL Deep Learning.

DNN Deep Neural Network.

FCM Fluorescence Confocal Microscopy.

FNNs Feedforward Neural Networks.

GANs Generative Adversarial Networks.

H&E Hematoxylin and Eosin stain.

LBP Local Binary Patterns.

ML Machine Learning.

MSE Mean Square Error.

NN Neural Network.

RCM Reflectance Confocal Microscopy.

ReLU Rectified Linear Unit.

SSIM Structural SIMilarity.

1 Introduction

1.1 Project background

In recent years, Deep Learning (DL) has significantly improved the performance of a wide range of computer vision tasks like image classification, object detection or semantic segmentation. GANs in particular have revolutionized generative tasks like image synthesis and image-to-image translation. Image-to-image translation is the task of generating an image based on a given source image with different characteristics depending on the specific problem.

In this work, based on Combalia et al. 2019 work in the Dermatology Department from the Hospital Clínic de Barcelona, the idea of image-to-image translation is applied to the transformation of Confocal Microscopy (CM) histological images into Hematoxylin and Eosin stain (H&E) appearance.

1.1.1 Confocal microscopy

CM is an optical imaging technique for increasing optical resolution and contrast of a micrograph by means of using a spatial pinhole to block out-of-focus light in image formation. With it, technicians are able to slice thin sections out of thick fluorescent specimens, view specimens in planes tilted to the line of sight, penetrate deep into light-scattering tissues, obtain 3D views at very high resolution... (Inoué 2006)

Ex vivo[1] confocal scanning laser microscopy can potentially accelerate Mohs surgery[2] by rapidly detecting carcinomas without conventional frozen histopathology (and its consequential time delays) (Chung et al. 2005).

Two different CM modes exist, RCM displays the backscattering signal of naturally occurring skin components, whereas Fluorescence Confocal Microscopy (FCM) provides contrast by using an applied fluorescent dye (Skvara et al. 2012). See figure 1.1 for an example.

[1] Ex vivo means that which takes place outside an organism. In science, ex vivo refers to experimentation or measurements done in or on tissue from an organism in an external environment with minimal alteration of natural conditions.

[2] Mohs micrographic surgery is considered the most effective technique for treating many basal cell carcinomas (BCCs) and squamous cell carcinomas (SCCs), the two most common types of skin cancer. The procedure is done in stages, including lab work, while the patient waits. This allows the removal of all cancerous cells for the highest cure rate while sparing healthy tissue and leaving the smallest possible scar.

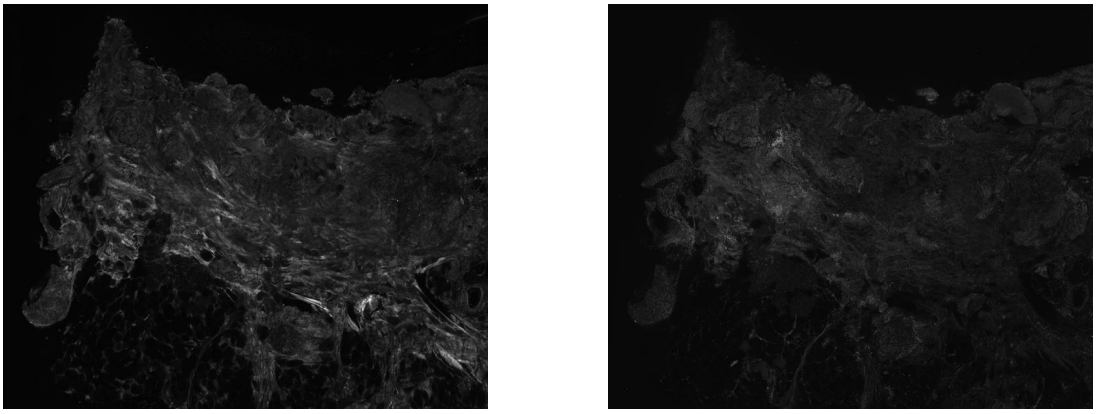


Figure 1.1: Example of a CM micrograph of a skin tissue. Reflectance mode on the left and fluorescence mode on the right

1.2 Problem statement

CM has enabled rapid evaluation of tissue samples directly in the surgery room significantly reducing the time of complex surgical operations in skin cancer (Cinotti et al. 2018), but the output largely differs from the standard H&E slides that pathologists typically use to analyze tissue samples. See figure 1.2b for a H&E example.

To bridge this gap, a method for combining the aforementioned modes of CM into a H&E-like image is presented in this work. A correctly done CM to H&E mapping would bring the efficiency of CM to untrained pathologists and surgeons.

Similar to a false color (also known as pseudo color) transformation, a parametric mapping function can be defined:

$$\mathbf{DSCM} = f_\theta(\mathbf{R}, \mathbf{F}) \tag{1.1}$$

where $\mathbf{DSCM} \in \mathbb{R}^{H \times W \times 3}$ (stands for digitally-stained CM) represents the resulting H&E-like RGB image, $\mathbf{R}, \mathbf{F} \in \mathbb{R}^{H \times W}$ represent the reflectance and fluorescence modes (respectively) of the CM input image with height H and width W.

1.2.1 Affine transformation

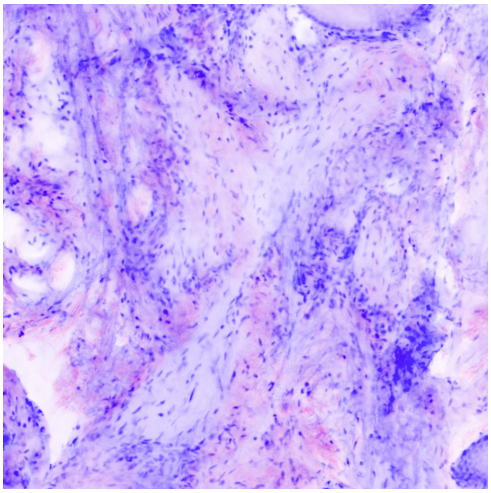
An affine transfomation is proposed in Gareau 2009 for the function f where the RGB values for each pixel are computed as:

$$\mathbf{DSCM}_{x,y} = \mathbf{1} - \mathrm{F}_{x,y}(\mathbf{1} - \mathbf{H}) - \mathrm{R}_{x,y}(\mathbf{1} - \mathbf{E}) \tag{1.2}$$

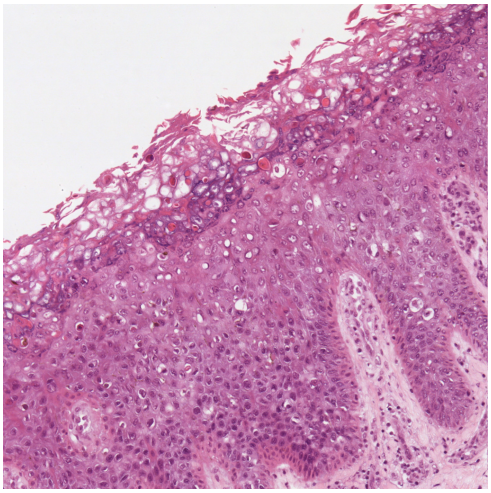
where:

$$\mathbf{H} = \begin{bmatrix} 0.30 & 0.20 & 1 \end{bmatrix}$$
$$\mathbf{E} = \begin{bmatrix} 1 & 0.55 & 0.88 \end{bmatrix}$$

These vectors represent coordinates in the RGB space (and). This way, the CM modes highlight different structures in distinct colours similar to a H&E slide; but as it can be seen in figure 1.2, the color scheme differs from an actual H&E sample.



(a) Digitally stained sample using the affine transformation



(b) H&E stained sample

Figure 1.2: Comparison between digital stain and H&E stain

1.2.2 Data-driven approach

In contrast to Gareau 2009 work where the parameters (**H** and **E**) are found experimentally, a data-driven approach of the problem will be taken where the parameters of the mapping function (1.1) are *learned* based on data. More specifically, the transformation will be defined by a Neural Network (NN) (called *StainNN*) and the parameters will be searched through an adversarial setting.

1.2.3 Speckle noise reduction

RCM images are affected by a multiplicative noise known as speckle, which may impair the performance of post-processing techniques. Hence, before digitally staining the CM images, this noise must be reduced.

The observed RCM image Y is related to the noise free image X by the following multiplicative model:

$$Y = X \odot F \tag{1.3}$$

Where $\odot$ denotes the Hadamard product (also known as the elementwise product) and F is the speckle noise random variable.

To reduce the speckle noise, a data-driven approach will also be taken using the called *DespecklingNN*.

1.3 Methods and procedures

This project was carried out at the Image and Video Processing Group (GPI) research group from the Signal Theory and Communications Department (TSC) at the Universitat Politecnica de Catalunya (UPC) in collaboration with the Dermatology Department from the Hospital Clínic de Barcelona.

The work presented in this thesis is the natural continuation of the work presented in Combalia et al. 2019.

1.4 Document structure

In section 2 an overview of relevant DL techniques and algorithms is presented to provide the reader a general knowledge of the field.

Section 3 contains the methodology of the project with detailed explanations of the models used to solve the presented problems. First the *DespeklingNN* is introduced in 3.2 then the *StainNN* in 3.3.

The experiments carried out to choose the right models are presented in section 4.

Finally, the conclusions and future work are discused in section 5.

2 Theoric background

2.1 Artificial Neural Networks

ANNs were originally developed as a mathematical model of the biological brain (McCulloch and Pitts 1943; Rosenblatt 1958; D. E. Rumelhart and McClelland 1987). Although ANNs have little resemblance to real biological neurons, they are a powerful Machine Learning (ML) tool and one of the most popular research topics in the last years. Nowadays, most researchers have shifted from the prespective of the biological neuron model to a more general *function approximator* point of view; in fact, it was proved that ANNs with enough capacity are capable of approximating any measurable function to any desired degree of accuracy (Cybenko 1989; Hornik 1991); this is, however, a non-constructive proof.

The basic structure of ANNs is a network of nodes (usually called neurons) joined to each other by weighted connections. Many varieties of ANNs have appeared over the years with different properties. One important distinction is between ANNs whose connections form feedback loops, and those whose connections are acyclic. ANNs with cycles are typically referred to as recurrent neural networks and those without cycles are known as Feedforward Neural Networks (FNNs).

In this work, only FNNs are used, in particular, a special kind that makes use of the convolution operation called Convolutional Neural Networks (CNNs). The following section provides an overview of this networks as well as the basic principles of training them.

2.1.1 Convolutional Neural Networks (CNN)

CNNs are a kind of ANNs particulary well-suited for computer vision tasks. They were first introduced in LeCun et al. 1998 to perform the task of hand-written digit classification and later popularized by Krizhevsky, Sutskever, and Geoffrey E Hinton 2012 entry on the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)[3], which won the classification task with a large margin of 10%.

CNNs consist of an input and an output layer, as well as multiple hidden layers (see figure 2.1). The input layer contains the data (e.g., RGB image) with minimal preprocessing (normalization, cropping...), in contrast to other ML algorithms that need

[3]ILSVRC is a competition to estimate the content of photographs for the purpose of retrieval and automatic annotation using a subset of the large hand-labeled ImageNet dataset (around 10,000,000 labeled images depicting 10,000+ object categories) as training. In the classification task, the algorithms are evaluated by the error rate in the test images presented with no initial annotation.

hand-engineered features. The output layer is different depending on the task.

Each hidden layer performs the convolution operation with one or more filters (commonly refered to as kernels by the DL community) taking the previous layer's output as the input and then an element-wise non-linear function is applied to the output. The non-linearities allow the model to extract hierarchical features (early layers extract the called low-level features and deeper layers extract high-level features) from the input data as it is illustrated in figure 2.2 on page 12 extracted from Matthew D Zeiler and Fergus 2014.

Down-sampling (also known as pooling in the DL literature) is also a very common operation applied after some hidden layers, aimed to make the model tranlation-invariant and reduce memory needs. Three main methods can be used to represent the set of N (or $N \times N$) neighbouring samples with a single number: *a*) Max-pooling uses the maximum value; *b*) The average value is used by the average-pooling method and *c*) Standard decimation "takes" a sample out of every N samples, it is usually implemented via N-strided (skipping $N-1$ positions when sliding the filter) convolution to compute only the used values.

Other kinds of layers like dropout (N. Srivastava et al. 2014) and batch-normalization (Ioffe and Szegedy 2015) can be used for regularization or faster training process.

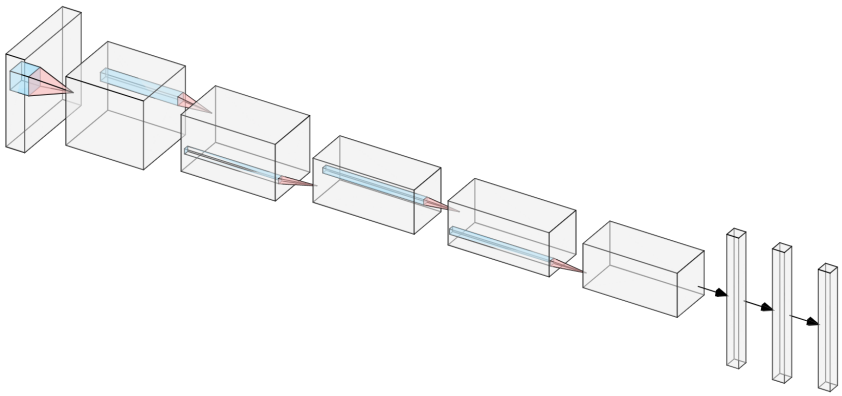


Figure 2.1: Visualization of a CNN with five convolutional layers and two fully-connected layers (a kind of layer not described in this work).

2.1.2 Training ANNs

Finding (or learning) the best set of parameters (θ) (weights, filters, ...) of a network for a given problem can be posed as an optimization problem by defining an appropiate objective function ($\mathcal{J}$). In a supervised setting, where training data composed by inputs

($\mathbf{X}$) and targets ($\mathbf{Y}$) is available:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathcal{J}\left((\mathbf{X}, \mathbf{Y}), f_\theta\right) \tag{2.1}$$

The complex structure of ANNs makes this optimization problem non-convex, hence iterative methods are used for moving towards an optimum solution; particulary, gradient descent based are the most common type as the networks are —by construction— fully-differentiable.

The idea behind gradient descent is simple. Given a (random) initial value for the input variables (e.g., filter coefficients in the case of CNNs), these are updated by moving towards the direction with greater slope i.e., the gradient. Moving towards the direction of the gradient (gradient ascent) will yield a local maximum, useful when maximizing the objective function; while moving toward the direction oposite to the gradient will yield a local minimum:

$$\theta_i^{(n)} \leftarrow \theta_i^{(n-1)} - \mu \frac{\partial \mathcal{J}}{\partial \theta_i}|_{\theta_i^{(n-1)}} \tag{2.2}$$

where the superscript (n) denotes the n-th iteration step and μ (known as learning rate) controls how much the variables should change between steps. Note that the length of the "step" is not only governed by the learning rate but also by the magnitute of the gradient.

In a DNN with thousands or millions of parameters, computing the gradient with respect to each parameter independently is computationally restrictive. In David E. Rumelhart, Geoffrey E. Hinton, and Williams 1986 work, an algorithm to efficiently train ANNs called backpropagation[4] which uses the principles of dynamic programming and exploits the chain rule was presented and it is how almost all ANNs are trained nowadays.

Cost function When training ANNs, the objective function is commonly defined as a cost function with two parts: the expected value of a loss plus a (weighted) regularization term.

$$\mathcal{J}\left((\mathbf{X}, \mathbf{Y}), f_\theta\right) = \mathbb{E}\{L\left(f_\theta(\mathbf{X}), \mathbf{Y}\right)\} + \lambda R(f_\theta) \tag{2.3}$$

$$\approx \frac{1}{N} \sum_{\mathbf{x}_i, \mathbf{y}_j \in \mathbf{X}, \mathbf{Y}} L\left(f_\theta(\mathbf{x}_i), \mathbf{y}_j\right) + \lambda R(f_\theta) \tag{2.3a}$$

[4]This lead to the terminology of forward pass, when the output of the model is computed sequentially from the input layer through each of the hidden layers, and the backward pass, when the partial derivatives a computed starting from the final layer and going back to the first hidden layer.

Equation (2.3a) shows how the expected value is approximated by taking the mean over N samples (batch size) of the dataset. Stochastic gradient descent is the case where $N = 1$, and mini-batch gradient descent when N is smaller than the total number of samples in the dataset; a small batch size is almost mandatory for large datasets, as computing the loss for every sample at each iteration step is very time and memory expensive and not only that: a small batch size helps avoiding bad local optima and improve generalization (Masters and Luschi 2018; Zhang et al. 2017).

The loss function will depend on our task. For example, in classification problems with c clases the cross entropy can be used: $L(\hat{\mathbf{y}}_i, \mathrm{y}_i) = -\log[\hat{\mathbf{y}}_i]_{\mathrm{y}_i}, \quad \mathrm{y}_i \in \{0 \,..\, c-1\}, \hat{\mathbf{y}}_i \in \mathbb{R}^c$; this loss enforces the model to make a good estimation of the class probabilities given an input datapoint. A loss function well aligned with our task is crucial, but defining such mathematical description of some problems is not always straightforward.

Advanced gradient-based optimization methods The basic (stochastic/mini-batch) gradient descent methods tend to find bad sub-optima when dealing with the noisy, non-convex landscape of ANNs; with a performance very sensitive to the initial values, learning rate and batch size. Many research work (Duchi, Hazan, and Singer 2011; G. Hinton, N. Srivastava, and Swersky 2012 (accessed September 14, 2019); Matthew D. Zeiler 2012; Diederik P. Kingma and Ba 2014) has focused on this area, developing algorithms that try to find better optima with fewer iterations. The one used in this work is the Adam "optimizer" (Diederik P. Kingma and Ba 2014) which defines the following update rule based on adaptive estimates of gradient moments:

$$\theta_i^{(n)} \leftarrow \theta_i^{(n-1)} - \mu \frac{\hat{m}_i^{(n)}}{\sqrt{\hat{v}_i^{(n)}} + \epsilon} \tag{2.4}$$

$$\hat{m}^{(n)} \leftarrow \frac{m^{(n)}}{1-\beta_1^n}, \quad m_i^{(n)} \leftarrow \beta_1 m_i^{(n-1)} + (1-\beta_1) g_i^{(n)} \tag{2.4a}$$

$$\hat{v}_i^{(n)} \leftarrow \frac{v_i^{(n)}}{1-\beta_2^n}, \quad v_i^{(n)} \leftarrow \beta_2 v_i^{(n-1)} + (1-\beta_2)\left(g_i^{(n)}\right)^2 \tag{2.4b}$$

$$g_i^{(n)} \leftarrow \frac{\partial \mathcal{J}}{\partial \theta_i}\Big|_{\theta_i^{(n-1)}} \tag{2.4c}$$

The parameter update equation (2.4) looks similar to the one in (2.2) but instead of directly using the gradient, its based on the exponential moving average of the gradient ($\hat{m}$) with a coefficient of $1-\beta_1$ (equation 2.4a) and the exponential moving average of the squared gradient ($\hat{v}$) with a coefficient of $1-\beta_2$ (equation 2.4b). The β's are refered to as momentum an typical values lie around 0.9. ϵ is a very small constant (e.g, 10^{-8})

to avoid division by zero. The intuition behind this update rule is that the steps are forced to be of size μ, as we are dividing by the magnitude of the gradient; and instead of taking the direction of the gradient at a given iteration step, the gradient value is passed through a low-pass filter to avoid making sudden changes that are common in stochatic optimization (specially with a small batch size).

2.2 Generative Adversarial Networks (GANs)

In ML generative models are ones which model the distribution of the data. This models can be used to perform classification through the conditional distribution (via a prior distribution of the classes) or to sample/generate data. Multiple generative models have been proposed that make use of DNN (G. E. Hinton 2009; Diederik P Kingma and Welling 2013; Goodfellow et al. 2014), GANs in particular have gained a lot of attention as they are capable of generating realistic images.

GANs is a framework for estimating generative models via an adversarial process, in which two models are trained: a generative model (G) that captures the data distribution, and a discriminative model (D).

Both models are implemented as DNN (usually CNNs in the case of image data). $G_{\theta_g}(\mathbf{z})$ maps from an input space of noise variables with distribution $p_z(\mathbf{z})$ to data space. $D_{\theta_d}(\mathbf{x})$ outputs the probability that $\mathbf{x}$ came from the data distribution $p_{data}(\mathbf{x})$ rather than G. See figure 2.3 for a simple illustration.

G is trained to minimize the likelihood of D assigning a low probability to its samples. While D is simultaneously trained to maximize the probability of assigning the correct label to both training examples and samples from G. Hence, the loss functions (see equation (2.3)) for each model are defined as (note how they go one against the other):

$$L_G = \log(1 - D(G(\mathbf{z}))) \tag{2.5a}$$

$$L_D = \begin{cases} -\log(D(\mathbf{x}), & \mathbf{x} \sim p_{data} \\ -\log(1 - D(\mathbf{x})) \equiv -\log(1 - D(G(\mathbf{z}))), & \mathbf{x} \sim p_g \end{cases} \tag{2.5b}$$

Equivalently, D and G play the following two-player minimax game:

$$\min_G \max_D \mathbb{E}_{x \sim p_{data}(x)}\{\log D(x)\} + \mathbb{E}_{z \sim p_z(z)}\{\log(1 - D(G(z)))\} \tag{2.6}$$

If G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{data}$. The discriminator will be unable to differentiate between the two distributions, i.e. $D(x) = 0.5$.

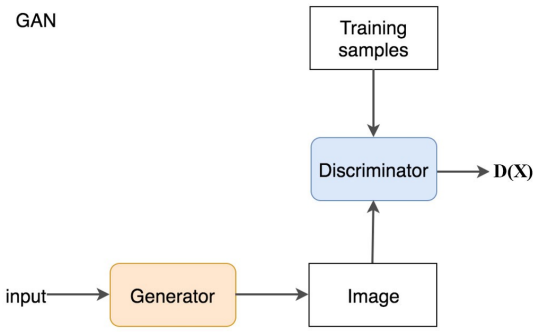


Figure 2.3: GANs training process diagram

This way of traning a generative model is unstable and can fail to converge due to a number of problems known as *failure modes*:

- Mode collapse: the generator collapses which produces limited varieties of samples,
- Diminished gradient: the discriminator gets too successful that the generator gradient vanishes and learns nothing,
- Unbalance between the generator and discriminator causing overfitting,

A number of publications (Arjovsky, Chintala, and Bottou 2017; Miyato et al. 2018; Salimans et al. 2016) tackle this problems with architecture and loss function modifications, and practical techniques. Successfuly tranined generators, generate very realistic samples as the constructed objective function essentially says "generate samples that look realistic".

2.2.1 GANs for image-to-image translation

In this section, two frameworks for image-to-image translation based on GANs are described. The first one, called pix2pix, (Isola et al. 2016) uses a conditional generative adversarial network to learn a mapping from input to output images using paired data. On the other hand, Cycle-Consistent Generative Adversarial Networks (CycleGANs) (Zhu et al. 2017) mapping is learned without paired data. Paired data consists of training examples $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{N}$ where correspondence between $\mathbf{x}_i$ and $\mathbf{y}_i$ exists, obtaining this kind of data can be difficult (or impossible) and expensive; contrarily, unpaired data consists of a source set $\mathbf{X}$ and a target set $\mathbf{Y}$ with no information provided as to which $\mathbf{x}_i$ matches which $\mathbf{y}_i$ (if any).

Pix2Pix Isola et al. 2016 work presents a conditional adversarial setting and applies it successfully to a variety of image-to-image translation problems that traditionally would require very different loss formulations; proving that *learned loss functions* are versatile. The losses used are similar to the ones in equations (2.5a) and (2.5b); but the discriminator not only gets the output of the generator as input, but the corresponing input as well (see figure 2.4 extracted from Isola et al. 2016). Apart from that, the generator is tasked to also be near the ground truth in a L1 sense; this is done by modifying the generator's loss function:

$$L_G = \log(1 - D(\mathbf{x}, G(\mathbf{x}))) + \lambda \left\| \mathbf{y} - G(\mathbf{x}) \right\|_1 \tag{2.7}$$

CycleGAN Zhu et al. 2017 presents a method that builds on top of the Pix2Pix framework for capturing special characteristics of one image collection and transfering these into another image collection in the absence of any paired training examples. In theory, an adversarially trained generator can learn to map images from a domain X to look indistiguishable from images from a domain Y; in practice, it is difficult to optimize the adversarial objective in isolation as this often leads to the mode collapse problem. In Zhu et al. 2017 work, this problem is addressed by adding more structure to the objective; concretely, it "encourages" the mapping to be cycle-consistent, i.e.: a function $G_{X \to Y}$ that maps from domain X to Y should have and inverse $G_{Y \to X}$ that maps its output to the original input; as in language translation, if a sentence is tranlated from Spanish to English and then back to Spanish we should arrive to a sentence close to the original. $G_{Y \to X}(G_{X \to Y}(\mathbf{x})) \approx \mathbf{x}$. This is done by simultaneously training two generators (with corresponding discriminators D_X, D_Y —notice how in this case, these do not take the source image as input—) and tasking them to not only "fool" their discriminator but to also produce an image that is close to the original input when translated back to the source domain (using the complementary generator), this is done by defining the following losses for the generators (more clearly visualized in figure 2.5).
An aditional term called idenity loss can be added to encourage the mapping to preserve color composition between the input and the output by making the generator be near an idenity mapping when samples from the target domain are provided. Note that the input and output domains need to have the same number of channels.
The final losses for the generators are the following:

$$\begin{aligned} L_{G_{X \to Y}} = &\log(1 - D_Y(G_{X \to Y}(\mathbf{x}))) \\ &+ \lambda_{cycle} \left\| \mathbf{x} - G_{Y \to X}(G_{X \to Y}(\mathbf{x})) \right\|_1 \\ &+ \lambda_{identity} \left\| \mathbf{y} - G_{X \to Y}(\mathbf{y}) \right\|_1 \end{aligned} \tag{2.8a}$$

$$\begin{aligned} L_{G_{Y \to X}} = &\log(1 - D_X(G_{Y \to X}(\mathbf{y}))) \\ &+ \lambda_{cycle} \left\| \mathbf{y} - G_{X \to Y}(G_{Y \to X}(\mathbf{y})) \right\|_1 \\ &+ \lambda_{identity} \left\| \mathbf{x} - G_{Y \to X}(\mathbf{x}) \right\|_1 \end{aligned} \tag{2.8b}$$

Note that neither Pix2Pix nor CycleGAN generators use a noise distribution to generate samples (in contrast to the original GANs framework).

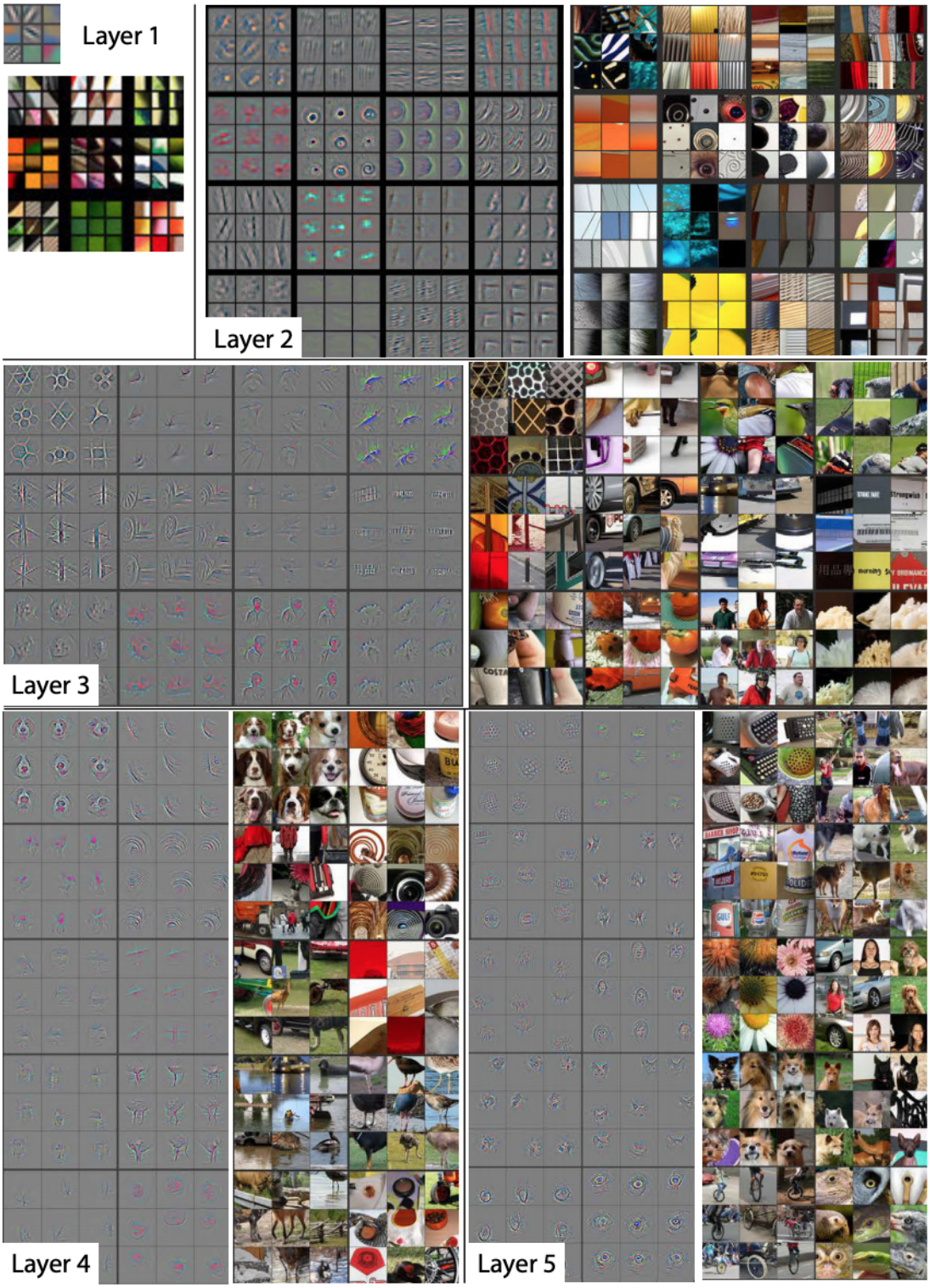


Figure 2.2: Visualization of features in a fully trained model. For layers 2-5 the top 9 activations in a random subset of feature maps are shown projected down to pixel space using the "deconvolutional" network introduced in Matthew D Zeiler and Fergus 2014 work

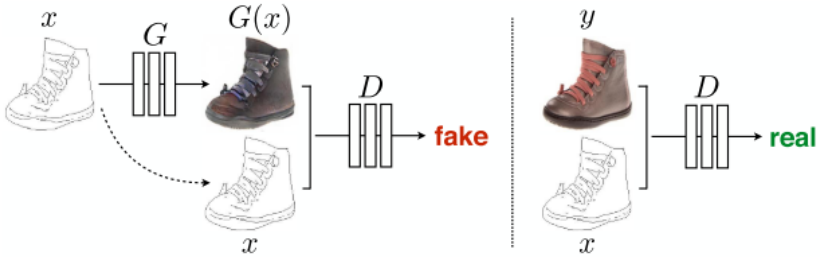


Figure 2.4: Training a conditional GAN to map edges to photo. The discriminator, D, learns to classify between fake and real edge, photo tuples. The generator, G, learns to fool the discriminator. Unlike an unconditional GAN, both the generator and discriminator observe the input edge map.

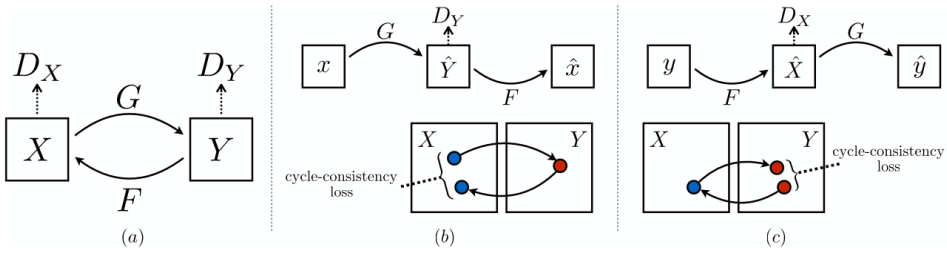


Figure 2.5: (Extracted from Zhu et al. 2017) The mapping model denoted as $G_{X \to Y}$ in this work is is denoted in this figure as G and $G_{Y \to X}$ as F

3 Methodology

3.1 Datasets

In this project two diferent datasets provided by the hospital clinic have been used to train and validate the despeckle and stain models: one for the CM domain and one for the H&E domain.

The Python implementation of all the datasets used can be found at appendix A.1

CM set The CM dataset consists of 27 large slides (around 100,000,000 pixels), each corresponing to a different sample of skin tissue. Some of this slides contain artifacts around the edges; in order to not provide this noise to the models, the slides are manually cropped avoiding this artifacts and focusing on the area containing the tissue. Working with such a large images is not practical since they do not fit in most GPUs' memory. A simple script to extract overlaping 1024x1024 patches out of the slides is developed so that, along with "on-the-fly" data augmentation techniques (random crop and random flip), more information can be extracted from the dataset. Only patches with a minimum mean grey level on both modes are extracted to be sure that they contain tissue texture. From that, a random sample of 1000 patches is used as a training set.

H&E set The H&E dataset contains a total of 560 crops of 1024x1024 pixels from whole slide histopathological images.

3.2 Despeckling network

As explained in section 1.2.3, RCM contain artifacts caused by a multiplicative noise (see figure 3.1 for an example). In this section different noise models are described and then the proposed methods for mitigating it are presented.

3.2.1 Speckle noise

As a means of having pairs of noisy-clean images needed to train a denoising model, FCM are artificially contaminated with a noise model. In SAR imaging, where speckle noise also appears, the noise is modeled by a gamma distribution with unit mean and variance $\frac{1}{L}$ (assuming the image is an average of L *looks*) (Ulaby and Dobson 1989):

$$F \sim \Gamma(k = L, \theta = \frac{1}{L}) \tag{3.1}$$

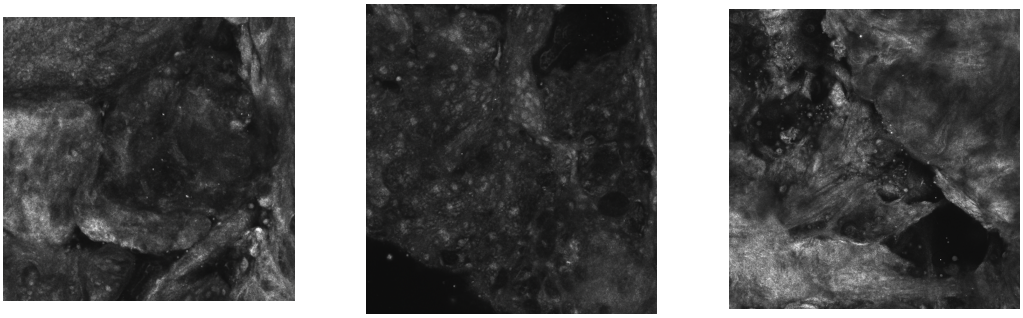


Figure 3.1: Three example of RCM images contaminated with speckle noise

Other models for the noise distribution exist; for instance, MATLAB's Image Processing Toolbox uses a uniform distribution with mean 1 and variance 0.05:

$$F \sim U(0.6535, 1.3464) \tag{3.2}$$

In the case of ultrasound imaging a rayleigh distribution with mean 1 is used (R. Srivastava and Gupta 2010):

$$F \sim Rayleigh(\sigma = \sqrt{\frac{2}{\pi}}) \tag{3.3}$$

Based on the appearence of artificially contaminated FCM images compared to naturally contaminated RCM, the experiments on this work are based on the gamma model (3.1).

3.2.2 Proposed network architectures

In order to filter the speckle noise, several DNN with the same basic structure are defined. Inspired by the ResNet (He et al. 2015a) they all share a *skip connection* between the first and last layer of a CNNs similar to P. Wang and Patel 2018 approach for SAR imaging despeckling.

The code for the PyTorch implemetations of all the models can be found at appendix A.3

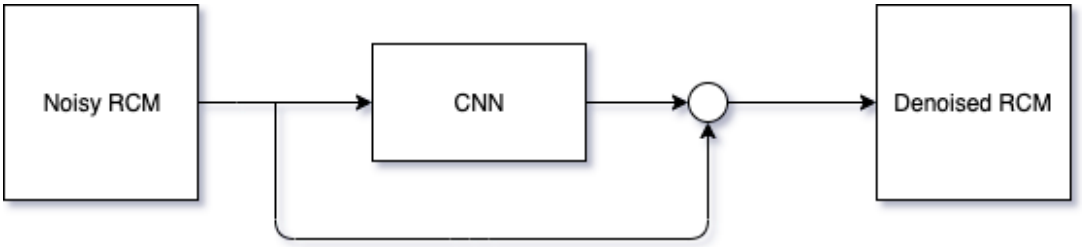


Figure 3.2: Skeleton of the despeckling networks. The node at the output of the CNN is the so-called residual connection, no operation is marked because the different variations are precisely defined by it.

The CNNs block consists of M convolutional layers each with K filters of size $N \times N$ and parametric Rectified Linear Unit (ReLU) activation functions (He et al. 2015b), except for the last layer which has one 1×1 filter to "merge" all the channels of the previous layer into a single channel image. Versions with and without an activation function are also defined.

The different model variations apply a distinct operation in the skip connection:

1. Division skip connection: A element-wise division between the input image and the network's output is defined, so that it makes a prediction of the noise. A priori this model seems prone to suffer an unstable training.

2. Multiplicative skip connection: A element-wise multiplication between the input image and the network's output is defined, so that it makes a prediction of the inverse of the noise. Although this estimation is more complicated, it is a "safer" alternative to the previous one.

3. Additive skip connection: A "classical" skip connection where matrix summation between the input and output of the network is performed. In this case, the operation is done in the log-space —the logarithm is applied to the input image and exponentiation to the model's output— in order to turn the noise into an additive one so is possible for the model to remove it.

3.3 Stain network

Obtaining aligned/paired data for CM to H&E is not possible since tissue blocks scanned with the CM need to undergo slicing before staining with H&E; hence, the staining models follow the CycleGANs framework introduced in section 2.2.1.

The discriminator model used is the PatchGAN (Zhu et al. 2017), the motivation behind this model is to model high-frequency "correctness"; this is done by classifying

(*real/fake*) small patches of the image (instead of the whole image) and averaging all the responses to privide the definitive output of the distriminator.

For the generator model a baseline is defined along with two families of advanced models.

3.3.1 Baseline

The baseline generator is a learned version of the affine transformation defined in (1.2). It is implemented as a single convolutional layer with 3 filters of size 1 so each output pixel's channel is a linear combination of the RCM and FCM values of the corresponding pixel of the source CM image plus a bias term.

3.3.2 Advanced models

Both families follow an encoder-decoder structure, i.e.: a series of convolution layers with down-sampling (encoder) followed by the same number of layers with up-sampling[5] (decoder), presumably the encoder maps the input into a latent representation where semantic transformations can be more easily defined and then the decoder "brings" it back to the image space.

Instead of directly using the CM modes, the digital staining method proposed in Gareau 2009 is used as source images for the geneators. The reason is twofold: on the one hand to provide more similar sets so that the mapping can be easier to learn, on the other hand, applying the identity loss demands for domains with equal number of channels.

1. ResNet-like generator: This model has the folling structure:

    - First layer: The input image is first mirror padded to mantain its dimensions, it is then convolved with 64 7×7 filters, the output is normalized with an instance normalization layer (Ulyanov, Vedaldi, and Lempitsky 2016) and followed by a ReLU activation function.

    - Down-sampling: This layer is composed by 3 2-strided (see down-sampling method *b*) in page 19) convolution layers with exponentially increasing number of kernels of shape 3×3 with instance normalization and ReLU after each layer. So after this block, the signal will have 512 channels and the height and width will be reduced by a factor of 8 (e.g., if the input image has size 256×256, the output will be 32×32).

    - Residual blocks: In a residual block, intead of trying to learn a transformation $\mathcal{T}(\mathbf{x})$, the residual $\mathcal{F}(\mathbf{x})$ is learnt so that $\mathcal{T}(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x}$ (ilustrated in figure

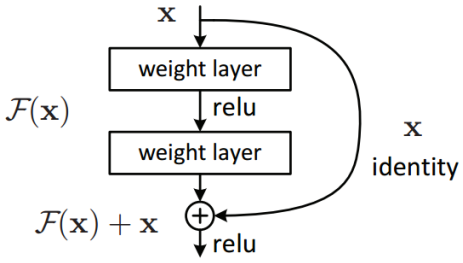


Figure 3.3: Residual block with 2 convolutional layers project. Note that $\mathbf{x}$ and $\mathcal{F}(\mathbf{x})$ should be of the same shape

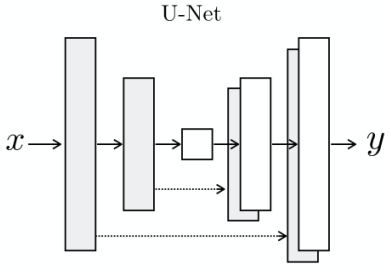


Figure 3.4: Skip connections between the UNet encoder and decorder

3.3); the motivation behind this is to avoid the problem know as the degradation problem where deeper ANNs perform worse than shallower counterparts. The generator network contains R two-layer residual blocks between the encoder and the decoder also with ReLU activation and instance normalization.

The last two blocks are the "mirror image" of the first two, i.e.:

- Up-sampling: 3 "up-convolution" layers with exponentialy decreasing number of filters so that the result is the same size as the original image.

- Final layer: Finally, the result of the previous layer is convolved by 3 7×7 filters and a tanh activation function is used to bound the output's range.

2. UNet-like generator: The UNet (Ronneberger, Fischer, and Brox 2015) is a fully-convolutional network originally designed for medical image segmentation, it differs from standard encoder-decoder network in how the decoder reconstructs the image from the latent representation:

   - Encoder: The encoder follows the same structure as the first two layers of the above described ResNet. No residual blocks are defined between the encoder and the decoder.

   - Decoder: As a means to obtain low-level information (location, texture, ...) from the encoder, the output from the corresponding encoder layer is concatenated to the output of the previous decoder layer (figure 3.4)

Both families use transposed convolutions[5] in the decoder network.

[5]Different methods for up-sampling can be used: the more classical interpolation methods (e.g., nearest neighbour or bicubic) or a *learnable* alternative called transposed convolution Noh, Hong, and Han 2015 (also known as fractionally strided convolution and sometimes wrongly refered to as deconvolution in some DL publications.). Chapter 4 of Dumoulin and Visin 2016 is suggested for readers not familiar with this operation.

3.4 Inference technique

Whole slide images are too large to fit directly on a GPU, therefore, the inference has to be tile-by-tile to obtain the stain transformed result. This introduces artifacts (see figure 3.5) between adjacent tiles in the output due to instance normalization relying on tile statistics. In order to fix this issue, the WSI inference technique from Bel et al. 2019 is applied.

So as to have neighbouring tiles with similar statistics, the method feeds overlapping regions to the model. The steps are the following (ilustrated in figure 3.6):

1. A large $N \times N$ patch (e.g. 2048×2048) is transformed.

2. The borders are cropped to obtain the center of half the size of the input: $\frac{N}{2} \times \frac{N}{2} = M \times M$.

3. The next prediction is made for a patch a quarter of the size apart from the previous one, i.e. the cropped output will have an overlap of 50% with the previous one.

4. The crops are combined by multiplying (element-wise) by a weight matrix and adding them. Two choices for this matrix are made: *a*) An "all 0.25" matrix: $\mathbf{W} = 0.25 * \mathbf{1}$ *b*) The outer-product of two translated triangular functions of length M: $[\mathbf{W}]_{m,n} = \Lambda(\frac{m}{M/2} - 1)\Lambda(\frac{n}{M/2} - 1)$

   A version with non-overlapping outputs is also developed where only the inputs overlap, instead of shifting the window by a quarter it is shifted by the half of the input. In this case the number of iterations to infer the whole slide is halved.

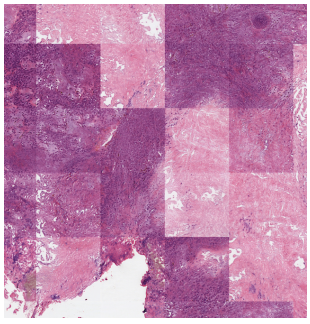


Figure 3.5: Tiling artifacts when infering whole slides tile-by-tile

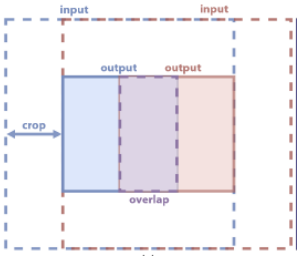


Figure 3.6: Inference is performed on a large input, after which half of the image is cropped. The window is shifted by quarter of the size, creating an overlap between tiles.

3.5 Quantitative measurement

Evaluating generator models is not straightforward, as metrics for image quality and diversity are dificult to define. Different methods are used for comparing methods: The inception score (IS) (Salimans et al. 2016) is used for measuring the quality of generated samples, the Fréchet Inception Distance is supposed to improve on the IS by comparing the statistics of generated samples to real samples. In this work, a texture descriptor is used to try to measure if the generated samples contain structures that are not present in the source image (popularly known as hallucinations). An example of an hallucination of a model during a failed training is shown in figure 3.7b.

The idea is to compare the input and output wholeslides by patches in a texture sense. This is done by computing a texture descriptor of the luminance of the source and the stained version and then computing a distance of the two. After trying various posibilities, the chosen texture descriptor is the Local Binary Patterns (LBP) histogram (Ojala, Pietikainen, and Maenpaa 2002).

LBP is a gray-level invariant feature extractor that assigns a number to every set of 9 neighboring pixels —a central pixel and its 8 closest pixels— in an image (in general, any number of pixels can be used but 9 is used here). The number is based on the difference of intensities between the neighbors and the central pixel: a 1 is assigned on pixels with a grey-level greater or equal than the center and 0 otherwise, this creates a code for each pixel in the image —by reading the assigned values clockwise starting at the top left corner as a binary word— that encodes the different posible edges. An histogram of this codes can be computed to obtain a description of the patterns that are found in a given image or patch.

The distance between the result and source patches is measured using the chi-squared distance between the normalized LBP histograms.

For reference purposes, the patch transformation depicted in figure 4.4 has a distance of 0.11.

(a) Linearly stained sample (b) Transformed sample with hallucinations

Figure 3.7: (a) is the input of the model that produces the output (b)

4 Experiments and results

4.1 GANs proof of concept

To test the adversarial setting versus a more traditional loss: MSE (equation 4.1), a "toy example" is carried out: train a generator model to fit a normal distribution with mean 0 and variance 1 (p_{data}).

The generator samples a vector of length 5 from a uniform ditribution (range -0.5-0.5) (p_z) and passes it through 3 fully connected layers[6] with LeakyReLU activation function and 5 units on each layer except for the final layer which only has 1 unit (it outputs a scalar) and no activation function (represented in figure 4.2).

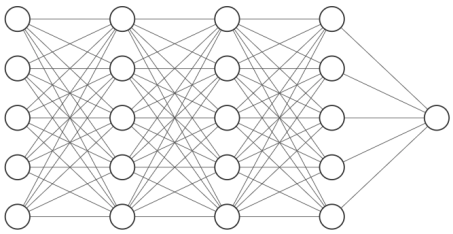


Figure 4.1: Generator architecture used for both the MSE and GANs setting

The target data comes from a sample of 100,000 i.i.d normal random variables. The models are trained for 100 epochs with a batch size of 100 using the Adam optimization algorithm presented in section 2.1.2 with a learning rate of 5×10^{-4} and default momentum values.

The results are compared by computing the Kolmogorov-Smirnov test of normality[7].

4.1.1 Results for MSE loss

As to be expected, the ANNs simply ignores the source of randomness (set first layer's weights close to zero) and produces an almost deterministic output close to the distribu-

[6] A fully connected layer outputs the matrix product between its input and a weight matrix and adds a bias vector, a node/neuron represents a row in the weight matrix plus the corresponding element in the bias vector: $\mathbf{y} = \mathbf{W}\mathbf{x}$, where $\mathbf{x}$ is the input represented by a column vector.

[7] The test statistic provides a measurement of the divergence of your sample distribution from the normal distribution. The higher the value of D, the less probable the data is normally distributed. The p-value quantifies this probability, with a low probability indicating that the sample diverges from a normal distribution to an extent unlikely to arise merely by chance. It is computed using the `https://www.socscistatistics.com/tests/kolmogorov` online tool.

tion's expected value: this is in fact the optimal solution for the MSE objective function.

$$L(G_\theta(\mathbf{x}), \mathbf{y}) = \|\mathbf{y} - G_\theta(\mathbf{x})\|_2^2 \tag{4.1}$$

Example of generated samples: `-0.05046 -0.05155 -0.05082 -0.05044`

The value of the K-S test statistic for 300 using this method is 0.1023. The p-value is 0.01819. This provides good evidence that the data is not normally distributed.

4.1.2 Results for adversarial loss

Rather than training G to minimize $\log(1-D(G(\mathbf{z})))$, G is trained to maximize $\log D(G(\mathbf{z}))$. This is a common practice as equation (2.5a) may not provide sufficient gradient for G to learn well early in training. It follows the same principle: try to fool the disctiminator.

$$L(G_\theta(\mathbf{x}), \mathbf{y}) = -\log D(G(\mathbf{z})) \tag{4.2}$$

The discriminator network is composed of 3 hidden layers with LeakyReLU as well, with a sigmoid activation function in the output layer.

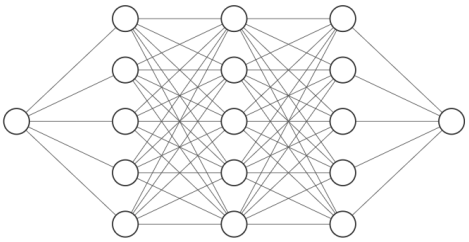


Figure 4.2: Discriminator architecture

In the case of the generator network traned in an adversarial manner, it produces samples with variability that are close to normally disributed (see figure 4.3 for a visual comparison with a true normal distribution).

The value of the K-S test statistic for 300 using this method is 0.03383. The p-value is 0.87045. This provides good evidence that the data does not differ significantly from that which is normally distributed.

This model clearly beats the MSE one on producing realistic and varied samples, but the training was much more unstable and the problem of mode collapse was encountered. Several training iterations with different hyperparameters (like number of layers, activation functions, learning rate,...) were necessary.

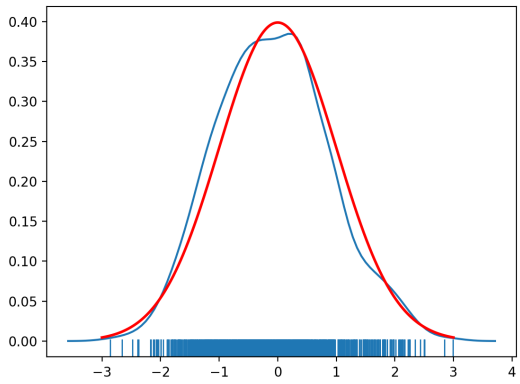


Figure 4.3: In blue a plot of the kernel density estimation using 1000 samples from the generator trained through the GANs framework. In red the probability density function of a normal random variable.

4.2 Despeckling network

The loss function used to define the objetive function the model will learn on is the MSE between FCM 256×256 crops and the artificially contaminated version of it.

$$L(f_\theta(\mathbf{x}_{noisy}), \mathbf{x}_{clean}) = \|\mathbf{x}_{clean} - f_\theta(\mathbf{x}_{noisy})\|_2^2 \tag{4.3a}$$

$$\mathbf{x}_{noisy} = \mathbf{x}_{clean} \odot \mathbf{s} \tag{4.3b}$$

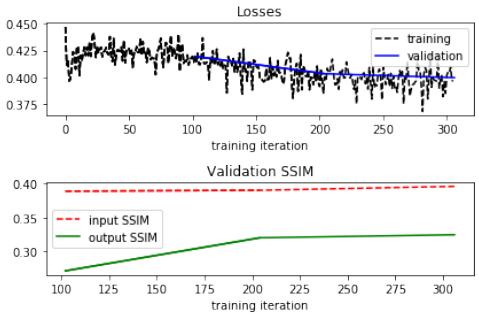
Where $\mathbf{s}$ is a realization of the speckle noise random variable model.

4.2.1 Model selection

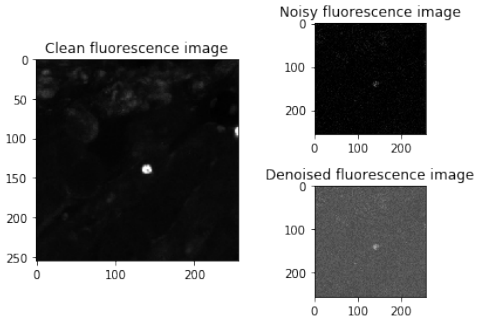
As explained in section 3.2, three different network architectures are implemented. To evaluate the model performance, a comparison of the Structural SIMilarity (SSIM) (Z. Wang et al. 2004) between noisy and clean (denoted $SSIM_{input}$); and denoised and clean (denoted $SSIM_{output}$) is used. Due to the lack of information about the confocal microcope used, the experiments are done with 2 different values $\{1, 5\}$ for the parameter L of the noise model (3.1).

The division skip-connection model is discarded early in the process because it fails to find any solution close to the desired.

In table 1 the mean SSIM values on the validation set for each of the final models are shown. The training process is done during 20 epochs using the Adam optimizer with a learning rate of 10^{-3} and a batch size of 32 samples.



(a) Learning curve or the division skip-connection model. $SSIM_{input}$ is greater than $SSIM_{output}$ throughout the trianing process



(b) Example of denoised FCM

Figure 4.4: The division skip-connection model fails to converge.

Model	M	K	N	mean $SSIM_{output}$
Multiply $(L = 1)$	3	32	5	0.877
Multiply $(L = 1)$	5	64	5	0.728
Multiply $(L = 5)$	5	64	5	0.947
Log-Add $(L = 1)$	3	32	5	0.960
Log-Add $(L = 1)$	5	64	5	0.806
Log-Add $(L = 5)$	5	64	5	0.965

Table 1: Models comparison with different number of layers M, number of filters K and filter size N. The validation set mean $SSIM_{input}$ is 0.414 for $L = 1$ and 0.723 for $L = 5$

4.3 Staining network

4.3.1 Baseline

The baseline presented in 3.3.1 is trained using the Adam optimizer with a batch size of 8, learning rate of 5×10^{-4}, momentums of 0.5 and 0.999 for β_1 and β_2 respectively. This hyperparameter values provide the best looking results, as with other parameters the training quickly destabilizes.

To provide an initialization closer to the optimal one, the weights are initializated with the weights defined by Gareau 2009 transformation.

The final results are very similar to the ones of the initial state, (see figure 4.5) so the model makes no real progress to make a better looking stain version.

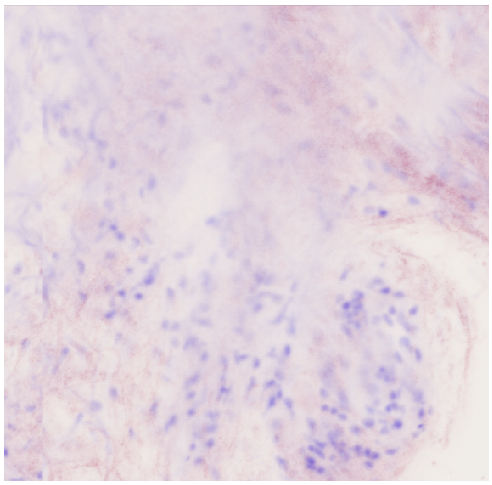


Figure 4.5: Example of a crop stained by the baseline model

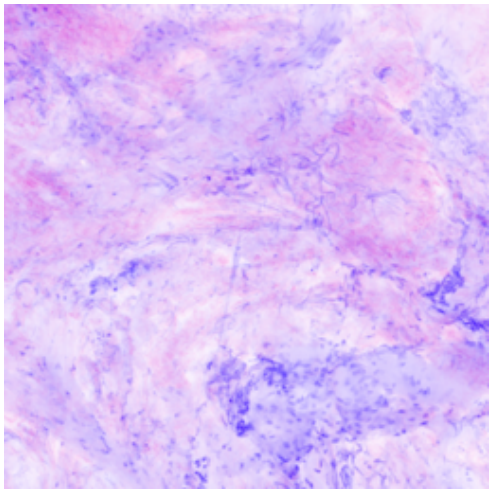


Figure 4.6

hyperparameter	value
λ_{cycle}	10
$\lambda_{identity}$	5
learning rate	2^{-4}
β_1	0.5
β_2	0.9
epochs	200
#layers D	3
residual blocks (residual model)	9
# down-sampling layers (UNet model)	7

Table 2: Table of hyperparameters used in staining network

4.3.2 Advanced DNN

The two architectures are trained using the hyperparameters described in table 2 for tand instead of optimizing the original GAN objective the so-called LSGAN —use the L2 distance instead of the cross-entropy— is used which is empirically shown to provide a more stable training (Mao et al. 2016).

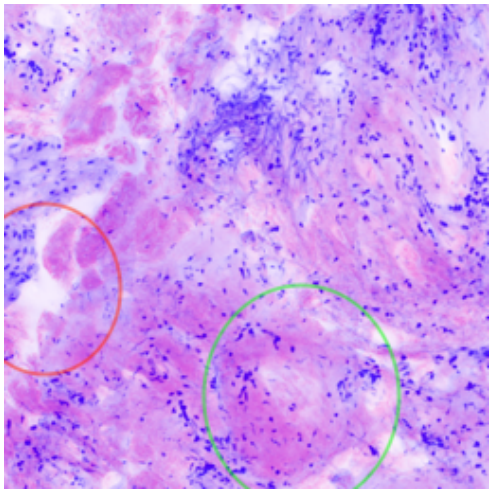
In theory the UNet-like model shoud mantain the structure and be less prone to "hallucinate", in practice this generally holds but still some structures are made up by the model; the residual model on the other hand sometimes eliminates nuclei present in the source image. Both cases can be seen in figure 4.7.

The mean value of the LBP histogram distance for the validation set is 0.0332 for the residual model and 0.0183 for the unet.

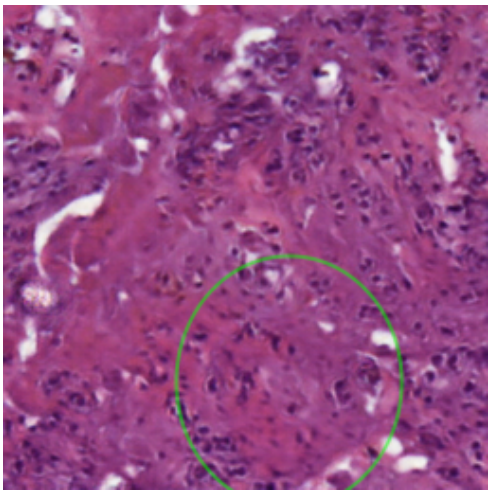
4.4 Inference method

The 2 of th inference methods described in section 3.4 are compared by computing the mean chi-squared distance of 512×512 patches in 7 large slides, i.e.: from a CM large mosaic, the digitally stained version by Gareau 2009 is computed (DSCM), the selected inference method is used to produce the H&E version using the U-Net model and the DSCM as input, then these are divided with a grid of 512×512 cells and the LBP histogram is computed on each cell, once the histograms are computed the chi-squared distance is used to measure how similar each cell is.

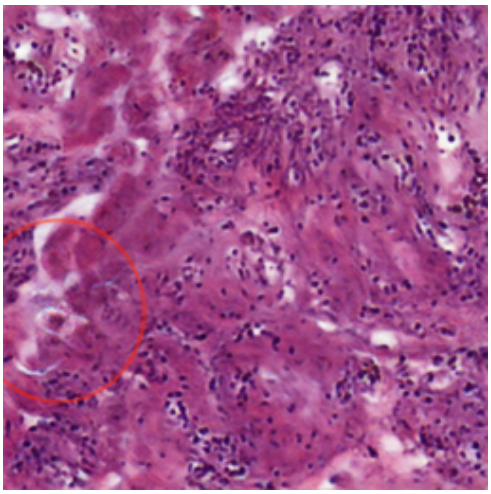
The plot in figure 4.8 shows how the method using 50% overlap and the weight metrix defined in 4.6 has a lower distribution of the metric for 7 different samples.



(a) Real



(b) Residual network



(c) UNet-like

Figure 4.7: (a) is the models' input, the second row show the outputs for the respective models. On (b) some nuclei that are in the input have been erased by the network —marked in green—. On (c) some structures are generated that are not present in the input —marked in red—

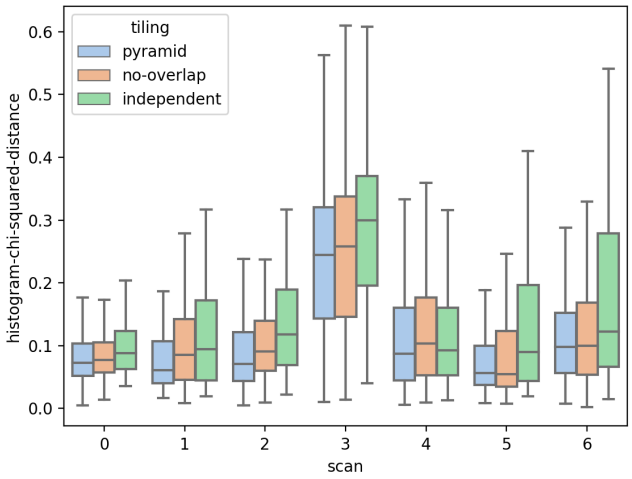


Figure 4.8: Independent tiling refers to inference made tile-by-tile with no overlap whatsoever. No-overlap refers to the technique where the inputs have overlap but the output is cropped so that they do not overlap. Pyramid is the WSI inference technique from Bel et al. 2019

5 Conclusions and future development

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus,

vehicula eu, lacus.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec odio elit, dictum in, hendrerit sit amet, egestas sed, leo. Praesent feugiat sapien aliquet odio. Integer vitae justo. Aliquam vestibulum fringilla lorem. Sed neque lectus, consectetuer at, consectetuer sed, eleifend ac, lectus. Nulla facilisi. Pellentesque eget lectus. Proin eu metus. Sed porttitor. In hac habitasse platea dictumst. Suspendisse eu lectus. Ut mi mi, lacinia sit amet, placerat et, mollis vitae, dui. Sed ante tellus, tristique ut, iaculis eu, malesuada ac, dui. Mauris nibh leo, facilisis non, adipiscing quis, ultrices a, dui.

References

Arjovsky, Martin, Soumith Chintala, and Léon Bottou (2017). "Wasserstein GAN". In: arXiv: 1701.07875. URL: http://arxiv.org/abs/1701.07875.

Bel, Thomas de et al. (2019). *Stain-Transforming Cycle-Consistent Generative Adversarial Networks for Improved Segmentation of Renal Histopathology*. Tech. rep., pp. 151–163. URL: http://proceedings.mlr.press/v102/de-bel19a/de-bel19a.pdf.

Chung, Vinh et al. (Jan. 2005). "Use of Ex Vivo Confocal Scanning Laser Microscopy during Mohs Surgery for Nonmelanoma Skin Cancers". In: *Dermatologic surgery : official publication for American Society for Dermatologic Surgery [et al.]* 30, pp. 1470–8. DOI: 10.1111/j.1524-4725.2004.30505.x.

Cinotti, Elisa et al. (Apr. 2018). "Ex vivo confocal microscopy: an emerging technique in dermatology". In: *Dermatology Practical & Conceptual* 8, pp. 109–119. DOI: 10.5826/dpc.0802a08.

Combalia, Marc et al. (2019). "Digitally Stained Confocal Microscopy through Deep Learning". In: *Proceedings of Machine Learning Research*, pp. 1–9.

Cybenko, G. (1989). "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4, pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

Duchi, John, Elad Hazan, and Yoram Singer (2011). *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*.

Dumoulin, Vincent and Francesco Visin (2016). *A guide to convolution arithmetic for deep learning*. arXiv: 1603.07285 [stat.ML].

Gareau, Daniel S. (2009). "Feasibility of digitally stained multimodal confocal mosaics to simulate histopathology". eng. In: *Journal of biomedical optics* 14.3, p. 34050. ISSN: 1083-3668. DOI: 10.1117/1.3149853. URL: https://www.ncbi.nlm.nih.gov/pubmed/19566342%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2929174/.

Goodfellow, Ian J et al. (2014). "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27*, pp. 1–9. ISSN: 10495258. DOI: 10.1016/B978-0-408-00109-0.50001-8. arXiv: arXiv:1406.2661v1. URL: https://arxiv.org/abs/1406.2661.

He, Kaiming et al. (2015a). *Deep Residual Learning for Image Recognition*. arXiv: 1512.03385 [cs.CV].

— (2015b). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. arXiv: 1502.01852 [cs.CV].

Hinton, G. E. (2009). "Deep belief networks". In: *Scholarpedia* 4.5. revision #91189, p. 5947. DOI: 10.4249/scholarpedia.5947.

Hinton, Geoffrey, Nitish Srivastava, and Kevin Swersky (2012 (accessed September 14, 2019)). *Overview of mini-batch gradient descent*. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Hornik, Kurt (1991). "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2, pp. 251–257. ISSN: 0893-6080. DOI: https://doi.org/10.

1016/0893-6080(91)90009-T. URL: http://www.sciencedirect.com/science/article/pii/089360809190009T.

Inoué, Shinya (2006). "Foundations of Confocal Scanned Imaging in Light Microscopy". In: *Handbook Of Biological Confocal Microscopy*. Ed. by James B. Pawley. Boston, MA: Springer US, pp. 1–19. ISBN: 978-0-387-45524-2. DOI: 10.1007/978-0-387-45524-2_1. URL: https://doi.org/10.1007/978-0-387-45524-2_1.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167. arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

Isola, Phillip et al. (Nov. 2016). "Image-to-Image Translation with Conditional Adversarial Networks". In: arXiv: 1611.07004. URL: http://arxiv.org/abs/1611.07004.

Kingma, Diederik P. and Jimmy Ba (2014). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG].

Kingma, Diederik P and Max Welling (2013). *Auto-Encoding Variational Bayes*. arXiv: 1312.6114 [stat.ML].

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. USA: Curran Associates Inc., pp. 1097–1105. URL: http://dl.acm.org/citation.cfm?id=2999134.2999257.

LeCun, Yann et al. (1998). "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Mao, Xudong et al. (2016). *Least Squares Generative Adversarial Networks*. arXiv: 1611.04076 [cs.CV].

Masters, Dominic and Carlo Luschi (2018). "Revisiting Small Batch Training for Deep Neural Networks". In: *CoRR* abs/1804.07612. arXiv: 1804.07612. URL: http://arxiv.org/abs/1804.07612.

McCulloch, Warren S. and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

Miyato, Takeru et al. (2018). "Spectral Normalization for Generative Adversarial Networks". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=B1QRgziT-.

Noh, Hyeonwoo, Seunghoon Hong, and Bohyung Han (2015). *Learning Deconvolution Network for Semantic Segmentation*. arXiv: 1505.04366 [cs.CV].

Ojala, T., M. Pietikainen, and T. Maenpaa (July 2002). "Multiresolution gray-scale and rotation invariant texture classification with local binary patterns". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.7, pp. 971–987. DOI: 10.1109/TPAMI.2002.1017623.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox (2015). "U-net: Convolutional networks for biomedical image segmentation". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. ISBN: 9783319245737. DOI: 10.1007/978-3-319-24574-4_28. arXiv: 1505.04597.

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

Rumelhart, D. E. and J. L. McClelland (1987). "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. MITP. URL: https://ieeexplore.ieee.org/document/6302929.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (Oct. 1986). "Learning representations by back-propagating errors". In: 323.6088, pp. 533–536. DOI: 10.1038/323533a0.

Salimans, Tim et al. (2016). "Improved Techniques for Training GANs". In: *CoRR* abs/1606.0. arXiv: 1606.03498. URL: http://arxiv.org/abs/1606.03498.

Skvara, Hans et al. (2012). "Combining in vivo reflectance with fluorescence confocal microscopy provides additive information on skin morphology". In: *Dermatology Practical & Conceptual*, pp. 3–12. DOI: 10.5826/dpc.0201a02.

Srivastava, Nitish et al. (2014). "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1, pp. 1929–1958.

Srivastava, Rajeev and J. R. P. Gupta (2010). "A PDE-Based Nonlinear Filter Adapted to Rayleigh's Speckle Noise for De-speckling 2D Ultrasound Images". In: *Contemporary Computing*. Ed. by Sanjay Ranka et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–12. ISBN: 978-3-642-14834-7.

Ulaby, Fawwaz T. (Fawwaz Tayssir) and M. Craig Dobson (1989). *Handbook of radar scattering statistics for terrain*. English. Includes index. Norwood, MA : Artech House. ISBN: 0890063362.

Ulyanov, Dmitry, Andrea Vedaldi, and Victor Lempitsky (2016). *Instance Normalization: The Missing Ingredient for Fast Stylization*. arXiv: 1607.08022 [cs.CV].

Wang, Puyang and Vishal M. Patel (2018). "Generating high quality visible images from SAR images using CNNs". In: *2018 IEEE Radar Conference, RadarConf 2018*, pp. 570–575. ISBN: 9781538641675. DOI: 10.1109/RADAR.2018.8378622.

Wang, Zhou et al. (2004). "Image quality assessment: from error visibility to structural similarity". In: *IEEE Transactions on Image Processing* 13.4, pp. 600–612. ISSN: 1057-7149 VO - 13. DOI: 10.1109/TIP.2003.819861. URL: http://www.cns.nyu.edu/pub/lcv/wang03-reprint.pdf.

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

Zeiler, Matthew D and Rob Fergus (2014). "Visualizing and understanding convolutional networks". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8689 LNCS. PART 1, pp. 818–833. ISBN: 9783319105895. DOI: 10.1007/978-3-319-10590-1_53. arXiv: arXiv:1311.2901v3.

Zhang, Chiyuan et al. (2017). "No . 067 April 4 , 2017 Theory of Deep Learning III : Generalization Properties of SGD by". In:

Zhu, Jun-Yan et al. (Mar. 2017). "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks". In: arXiv: 1703.10593. URL: http://arxiv.org/abs/1703.10593.

Appendices

A PyTorch implementations

A.1 Datasets

At the heart of PyTorch data loading utility is the `torch.utils.data.DataLoader` class. It represents a Python iterable over a dataset. The most important argument of DataLoader constructor is dataset, which indicates a dataset object to load data from. PyTorch supports two different types of datasets:

- Map-style datasets,
- Iterable-style datasets.

In this project all datasets are map-style. A map-style dataset is one that implements the `__getitem__()` and `__len__()` protocols, and represents a map from (possibly non-integral) indices/keys to data samples.

```
import os
import pathlib
from abc import ABCMeta, abstractmethod
import warnings

import torch.utils.data
import pyvips
from PIL import Image
import openslide

from transforms import VirtualStainer, MultiplicativeNoise


def return_prefix_decorator(getitem):
    """Used to wrap __getitem__ method.

    If return_prefix attribute is True, it will make
    __getitem__ return the desired item along with its file
    prefix using _get_prefix method
    (this can be used to identify the samples).
    """
    def getitem_wrapper(self, item):
        sample = getitem(self, item)
        if self.return_prefix:
            prefix = self.get_prefix(item)
```

```
            return sample, prefix
        else:
            return sample
    return getitem_wrapper


class CMDataset(torch.utils.data.Dataset, metaclass=ABCMeta):
    """CM scans dataset abstract class with possibility
    to (linearly) stain."""
    def __init__(self, transform=None,
                 only_R=False, only_F=False, stain=False,
                 transform_stained=None,
                 transform_F=None, transform_R=None,
                 return_prefix=False):
        """
        Args:
            only_R (bool): return only R mode.
            only_F (bool): return only F mode.
                If both only_R and only_F are True,
                the former takes precedence.
            stain (bool): Stain CM image using VirtualStainer.
            transform_stained: Apply transform to stained image.
            transform_F: Apply transform to F-mode image.
            transform_R: Apply transform to R-mode image.
            transform (callable): Apply transform to both modes
                (after respective transforms).
                R and F modes will be used as argument
                in that order.
        """
        if only_R and only_F:
            raise ValueError("Only one (if any) of 'only'"
                             "options must be true.")
        self.only_R, self.only_F = only_R, only_F
        self.transform_stained = transform_stained
        self.transform_F = transform_F
        self.transform_R = transform_R
        self.transform = transform
        self.scans = self._list_scans()
        self.stainer = VirtualStainer() if stain else None
        self.return_prefix = return_prefix

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

    @abstractmethod
    def get_f(self, item):
```

```
        """Return item-th sample F mode."""
        pass

    @abstractmethod
    def get_r(self, item):
        """Return item-th sample R mode."""
        pass

    @abstractmethod
    def get_prefix(self, item):
        """Return item-th sample prefix."""
        pass

    @abstractmethod
    def _list_scans(self):
        pass

    @return_prefix_decorator
    def __getitem__(self, item):
        """Get CM image.

        If stain, return stained image
        (using transforms.VirtualStainer).
        Return both modes otherwise.
        If return_prefix, return (sample, prefix) tuple.
        """
        # load R mode if needed
        r_img = None if self.only_F else self.get_r(item)
        # load F mode if needed
        f_img = None if self.only_R else self.get_f(item)

        if self.transform_F:
            f_img = self.transform_F(f_img)
        if self.transform_R:
            r_img = self.transform_R(r_img)

        if self.transform:
            r_img, f_img = self.transform(r_img, f_img)

        if self.stainer:
            img = self.stainer(r_img, f_img)
            if self.transform_stained:
                return self.transform_stained(img)
            return img

        if self.only_R:
```

```
            return r_img
        elif self.only_F:
            return f_img
        return {'F': f_img, 'R': r_img}


class ColonCMDataset(CMDataset):
    """CM colon scans dataset with possibility to stain.

    785: R
    488: F
    """

    def __init__(self, root_dir, **kwargs):
        self.root_dir = pathlib.Path(root_dir)
        super().__init__(**kwargs)

    def _list_scans(self):
        scans_R = list(self.root_dir.glob('**/785.png'))
        scans_F = list(self.root_dir.glob('**/488.png'))
        assert len(scans_F) == len(scans_R)
        # list of (R,F) pairs, needs to be list so it has len().
        scans = list(zip(sorted(scans_R), sorted(scans_F)))

        return scans

    def get_f(self, item):
        # second element of tuple is F mode
        f_file = self.scans[item][1]
        f_img = pyvips.Image.new_from_file(str(f_file))
        return f_img

    def get_r(self, item):
        # first element of tuple is R mode
        r_file = self.scans[item][0]
        r_img = pyvips.Image.new_from_file(str(r_file))
        return r_img

    def get_prefix(self, item):
        return self.scans[item][0][:-8]


class ColonHEDataset(torch.utils.data.Dataset):
    """H&E colon scans dataset."""

    def __init__(self, root_dir, transform=None, alpha=False):
```

```
        """
        Args:
            root_dir (str): Directory with mosaic directories.
            transform (callable): Apply transform to image.
            alpha (bool): return slide with alpha channel.
        """
        self.root_dir = pathlib.Path(root_dir)
        self.transform = transform
        self.alpha = alpha
        self.scans = sorted(list(self.root_dir.glob('*.bif')))

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

    def __getitem__(self, item):
        """Get max resolution H&E image.

        :return openslide.OpenSlide object
        """
        scan = openslide.OpenSlide(str(self.scans[item]))
        if self.alpha:
            return scan
        if self.transform:
            scan = self.transform(scan)
        return scan


class SkinCMDataset(CMDataset):
    """CM skin scans dataset with possibility to stain.

    DET#1: R
    DET#2: F
    """

    def __init__(self, root_dir, **kwargs):
        self.root_dir = root_dir
        super().__init__(**kwargs)

    def _list_scans(self):
        scans = []
        for root, dirs, files in os.walk(self.root_dir):
            if 'mosaic' in root.split('/')[-1]:
                scans.append(root)

        scans = sorted(list(set(scans)))
        return scans
```

```
    def get_f(self, item):
        f_file = self.scans[item] + '/DET#2/highres_raw.tif'
        f_img = pyvips.Image.new_from_file(
                        f_file, access='random')
        return f_img

    def get_r(self, item):
        r_file = self.scans[item] + '/DET#1/highres_raw.tif'
        r_img = pyvips.Image.new_from_file(
                        r_file, access='random')
        return r_img

    def get_prefix(self, item):
        return self.scans[item]


class CMCropsDataset(CMDataset):
    """CM scans crops dataset with possibility to stain.

    To extract crops from wholeslides use save_crops.py script.
    """

    def __init__(self, root_dir, **kwargs):
        self.root_dir = pathlib.Path(root_dir)
        super().__init__(**kwargs)

    def _list_scans(self):
        crops_R = {str(r)[:-6]
                   for r in self.root_dir.glob('*R.tif')}
        crops_F = {str(f)[:-6]
                   for f in self.root_dir.glob('*F.tif')}
        if self.only_R:
            crops = crops_R
        elif self.only_F:
            crops = crops_F
        else:
            # if use both modes,
            # use only the crops with both modes available.
            if len(crops_F) != len(crops_R):
                warnings.warn(
                        'Number of crops for R and F modes '
                        'are different. Dataset will be only '
                        'composed by the images with '
                        'both modes available.')
            crops = crops_R & crops_F  # set intersection.
```

```
        return sorted(crops)

    def get_f(self, item):
        f_file = self.scans[item] + '_F.tif'
        f_img = Image.open(f_file)
        return f_img

    def get_r(self, item):
        r_file = self.scans[item] + '_R.tif'
        r_img = Image.open(r_file)
        return r_img

    def get_prefix(self, item):
        return os.path.basename(self.scans[item])


class NoisyCMCropsDataset(CMCropsDataset):
    """Dataset with 512x512 CM crops with speckle noise."""

    def __init__(self, root_dir, mode, noise_args,
                 transform=None, return_prefix=False):
        """

        :param root_dir: Directory with "mosaic" directories.
        :param mode: which mode to work with (F or R).
        :param noise_args: dict with random_variable and
                parameter keys.
        """
        if mode == 'F':
            super().__init__(
                root_dir, only_F=True, transform_F=transform,
                return_prefix=return_prefix)
        elif mode == 'R':
            super().__init__(
                root_dir, only_R=True, transform_R=transform,
                return_prefix=return_prefix)
        else:
            raise ValueError(
                "'mode' parameter should be 'F' or 'R'")
        self.add_noise = MultiplicativeNoise(**noise_args)

    def __getitem__(self, item):
        """Return (noisy, clean) tuple.

        If return_prefix, return ((noisy, clean), prefix)
        Return (noisy, clean) otherwise.
```

```
        """
        clean = super().__getitem__(item)
        if self.return_prefix:
            clean, prefix = clean
        noisy = self.add_noise(clean)
        if self.return_prefix:
            return (noisy, clean), prefix
        return noisy, clean


class SimpleDataset(torch.utils.data.Dataset):
    EXTENSIONS = ['.png', '.jpg', '.tif']

    def __init__(self, root_dir, transform=None,
                 return_prefix=False):
        self.root_dir = pathlib.Path(root_dir)
        self.files = [file for file in self.root_dir.glob('*.*')
                      if file.suffix in self.EXTENSIONS]

        self.transform = transform
        self.return_prefix = return_prefix

    @return_prefix_decorator
    def __getitem__(self, item):
        img = Image.open(self.files[item])
        if self.transform:
            img = self.transform(img)
        return img

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

    def get_prefix(self, item):
        return os.path.basename(self.files[item])

class UnalignedCM2HEDataset(torch.utils.data.Dataset):
    def __init__(self, cm_root, he_root,
        transform_cm=None, transform_he=None):
        self.cm_dataset = CMCropsDataset(
                cm_root, transform=transform_cm)
        self.he_dataset = SimpleDataset(
                he_root, transform=transform_he)

        self.cm_to_tensor = CMToTensor()
        self.he_to_tensor = torchvision.transforms.ToTensor()
```

```
    def __getitem__(self, item):
        cm = self.cm_dataset[item % len(self.cm_dataset)]
        cm = self.cm_to_tensor(cm['R'], cm['F'])
        he = self.he_dataset[
                random.randrange(len(self.he_dataset))]
        he = self.he_to_tensor(he)

        return {'CM': cm, 'HE': he}

    def __len__(self):
        return max(len(self.cm_dataset), len(self.he_dataset))
```

A.2 Transforms

Some useful object oriented transformations are defined in a similar way to the ones defined in `torchvision.transforms` package.

```
import random

import pyvips
import numpy as np
import torch
import torchvision.transforms.functional as TF


class VirtualStainer:
    """Class for digitally staining CM to H&E histology
    using Daniel S. Gareau technique."""

    H = [0.30, 0.20, 1]
    one_minus_H = list(map(lambda x: 1 - x, H))
    E = [1, 0.55, 0.88]
    one_minus_E = list(map(lambda x: 1 - x, E))

    def __call__(self, sample_R, sample_F):
        """Apply staining transformation and return pyvips image.

        sample_R: pyvips.Image or numpy array with range [0,1]
        sample_F: pyvips.Image or numpy array with range [0,1]
        """
        if (isinstance(sample_F, pyvips.Image)
                and isinstance(sample_R, pyvips.Image)):
            f_res = sample_F * self.one_minus_H
            r_res = sample_R * self.one_minus_E
```

```
            image = 1 - f_res - r_res
            res = image.copy(
                interpretation=pyvips.enums.Interpretation.RGB)
            return res

        # assumes sample_F and sample_R are numpy arrays
        f_res = sample_F * np.array(
                self.one_minus_H).reshape((3, 1, 1))
        r_res = sample_R * np.array(
                self.one_minus_E).reshape((3, 1, 1))

        return 1 - f_res - r_res


class MultiplicativeNoise:
    """Multiply by random variable."""

    def __init__(self, random_variable, **parameters):
        """

        random_variable: numpy.random distribution function.
        """
        self.random_variable = random_variable
        self.parameters = parameters

    def __call__(self, img):
        """return clean image and contaminated image."""
        noise = torch.tensor(
            self.random_variable(size=img.size(),
                                 **self.parameters),
            device=img.device, dtype=img.dtype,
            requires_grad=False
        )
        return img * noise, img


class CMMinMaxNormalizer:
    """Min-max normalize CM sample with different methods.

    Independent method "min-max" normalizes each mode
            separately.
    Global method "min-max" normalizes with global min and
            max values.
    Average method "min-max" normalizes with min and max
            values of the average image.
```

```
    """

    def __init__(self, method):
        assert method in ('independent', 'global', 'average')
        self.method = method

    def __call__(self, sample_R, sample_F):
        if self.method == 'independent':
            new_R = self._normalize(sample_R)
            new_F = self._normalize(sample_F)
        elif self.method == 'global':
            # compute min and max values.
            min_R, max_R = sample_R.min(), sample_R.max()
            min_F, max_F = sample_F.min(), sample_F.max()
            # get global min and max.
            min_ = min_R if min_R > min_F else min_F
            max_ = max_R if max_R > max_F else max_F
            # normalize with global min and max.
            new_R = self._normalize(sample_R, min_, max_)
            new_F = self._normalize(sample_F, min_, max_)
        else:  # self.method == average
            avg = (sample_R + sample_F) / 2
            min_ = avg.min()
            max_ = avg.max()
            new_R = self._normalize(sample_R, min_, max_)
            new_F = self._normalize(sample_F, min_, max_)
        return new_R, new_F

    @staticmethod
    def _normalize(img, min_=None, max_=None):
        """Normalize pyvips.Image by min and max."""
        if min_ is None:
            min_ = img.min()
        if max_ is None:
            max_ = img.max()
        return (img - min_) / (max_ - min_)


class CMRandomCrop:

    def __init__(self, height, width):
        self.height = height
        self.width = width

    def __call__(self, R, F):
        r_height, r_width = R.size
```

```
        f_height, f_width = F.size
        assert r_height == f_height
        assert r_width == f_width
        rand_i = random.randrange(r_height - self.height)
        rand_j = random.randrange(r_width - self.width)

        R = TF.crop(R, rand_i, rand_j,
                    self.height, self.width)
        F = TF.crop(F, rand_i, rand_j,
                    self.height, self.width)
        return R, F


class CMRandomHorizontalFlip:

    def __call__(self, R, F):
        if random.random() > 0.5:
            R = TF.hflip(R)
            F = TF.hflip(F)
        return R, F


class CMRandomVerticalFlip:

    def __call__(self, R, F):
        if random.random() > 0.5:
            R = TF.vflip(R)
            F = TF.vflip(F)
        return R, F


class CMToTensor:

    def __call__(self, R, F):
        R = TF.to_tensor(R)
        F = TF.to_tensor(F)
        return torch.cat((R, F))


class CMCompose:
    """Composes several transforms together."""

    def __init__(self, transforms):
        self.transforms = transforms

    def __call__(self, R, F):
```

```
        for t in self.transforms:
            R, F = t(R, F)
        return R, F
```

A.3 Models

PyTorch models are implemented by subclassing the `torch.nn.Module` abstract class which defines the abstract method `forward` that should implement the forward pass of the model. The backward pass is performed by a `torch.optim.Optimizer` subclasses which make use of the PyTorch's automatic differentiation system "autograd". `torch.nnModule` subclasses can in turn contain other `torch.nn.Module` objects, this is how layers are usually defined.

Despeckling models

```
import torch.nn as nn
import torch

SAFE_LOG_EPSILON = 1E-5  # small number to avoid log(0).
SAFE_DIV_EPSILON = 1E-8  # small number to avoid division by zero.


class ResModel(nn.Module):
    """Model with residual/skip connection."""

    def __init__(self, sub_module,
                 skip_connection=lambda x, y: x + y):
        """

        :param sub_module: model between input and
                skip connection.
        :param skip_connection: operation to do in
                skip connection.
        """
        super(ResModel, self).__init__()

        self.skip_connection = skip_connection

        self.noise_removal_block = sub_module

    def forward(self, x):
        clean = self.skip_connection(
```

```
                x, self.noise_removal_block(x))

        return clean


class BasicConv(nn.Module):
    """Series of convolution layers keeping the
    same image shape."""

    def __init__(self, in_channels=1, n_layers=6,
                 n_filters=64, kernel_size=3):
        super(BasicConv, self).__init__()
        model = [nn.Sequential(nn.Conv2d(in_channels,
                                         n_filters,
                                         kernel_size,
                                         padding=kernel_size // 2),
                               nn.BatchNorm2d(n_filters),
                               nn.PReLU())
                 ]
        for _ in range(n_layers - 1):
            model += [nn.Sequential(nn.Conv2d(n_filters,
                                              n_filters,
                                              kernel_size,
                                              kernel_size // 2),
                                    nn.BatchNorm2d(n_filters),
                                    nn.PReLU())
                      ]
        model += [nn.Conv2d(n_filters, in_channels, 1)]
        self.model = nn.Sequential(*model)

    def forward(self, x):
        return self.model(x)


class DilatedConv(nn.Module):
    """Series of convolution layers with dilation
    2 keeping the same image shape."""

    def __init__(self, in_channels=1, n_layers=6,
                 n_filters=64, kernel_size=3):
        super(DilatedConv, self).__init__()
        model = [nn.Sequential(nn.Conv2d(in_channels,
                                         n_filters,
                                         kernel_size,
                                         kernel_size // 2 * 2,
                                         dilation=2),
```

```
                                    nn.BatchNorm2d(n_filters),
                                    nn.PReLU())
                    ]
        for _ in range(n_layers - 1):
            model += [nn.Sequential(nn.Conv2d(
                                            n_filters,
                                            n_filters,
                                            kernel_size,
                                            kernel_size // 2 * 2,
                                            dilation=2),
                                    nn.BatchNorm2d(n_filters),
                                    nn.PReLU())
                      ]
        model += [nn.Conv2d(n_filters, in_channels, 1)]
        self.model = nn.Sequential(*model)

    def forward(self, x):
        return self.model(x)


class LogAddDespeckle(nn.Module):
    """Apply log to pixel values, residual block with
    addition, apply exponential."""

    def __init__(self, n_layers=6, n_filters=64,
                 kernel_size=3, apply_sigmoid=True):
        super(LogAddDespeckle, self).__init__()
        conv = BasicConv(in_channels=1, n_layers=n_layers,
                         n_filters=n_filters,
                         kernel_size=kernel_size)
        self.remove_noise = ResModel(
            conv, skip_connection=lambda x, y: x + y)
        self.apply_sigmoid = apply_sigmoid

    def forward(self, x):
        log_x = (x + SAFE_LOG_EPSILON).log()
        clean_log_x = self.remove_noise(log_x)
        clean_x = clean_log_x.exp()
        if self.apply_sigmoid:
            return torch.sigmoid(clean_x)
        return clean_x


class DilatedLogAddDespeckle(nn.Module):
    """Apply log to pixel values, residual block with addition,
    apply exponential."""
```

```
    def __init__(self, n_layers=6, n_filters=64,
                 kernel_size=3, apply_sigmoid=True):
        super(DilatedLogAddDespeckle, self).__init__()
        conv = DilatedConv(in_channels=1, n_layers=n_layers,
                           n_filters=n_filters,
                           kernel_size=kernel_size)
        self.remove_noise = ResModel(
                conv, skip_connection=lambda x, y: x + y)
        self.apply_sigmoid = apply_sigmoid

    def forward(self, x):
        log_x = (x + SAFE_LOG_EPSILON).log()
        clean_log_x = self.remove_noise(log_x)
        clean_x = clean_log_x.exp()
        if self.apply_sigmoid:
            return torch.sigmoid(clean_x)
        return clean_x


class LogSubtractDespeckle(nn.Module):
    """Apply log to pixel values, residual block with
    subtraction, apply exponential."""

    def __init__(self, n_layers=6, n_filters=64,
                 kernel_size=3, apply_sigmoid=True):
        super(LogSubtractDespeckle, self).__init__()
        conv = BasicConv(in_channels=1, n_layers=n_layers,
                         n_filters=n_filters,
                         kernel_size=kernel_size)
        self.remove_noise = ResModel(
            conv, skip_connection=lambda x, y: x - y)
        self.apply_sigmoid = apply_sigmoid

    def forward(self, x):
        log_x = (x + SAFE_LOG_EPSILON).log()
        clean_log_x = self.remove_noise(log_x)
        clean_x = clean_log_x.exp()
        if self.apply_sigmoid:
            return torch.sigmoid(clean_x)
        return clean_x


class MultiplyDespeckle(nn.Module):
    """Residual block with multiplication."""
```

```
    def __init__(self, n_layers=6, n_filters=64,
                 kernel_size=3, apply_sigmoid=True):
        super(MultiplyDespeckle, self).__init__()
        conv = BasicConv(in_channels=1, n_layers=n_layers,
                         n_filters=n_filters,
                         kernel_size=kernel_size)
        self.remove_noise = ResModel(
                conv, skip_connection=lambda x, y: x * y)
        self.apply_sigmoid = apply_sigmoid

    def forward(self, x):
        clean_x = self.remove_noise(x)
        if self.apply_sigmoid:
            return torch.sigmoid(clean_x)
        return clean_x


class DivideDespeckle(nn.Module):
    """Residual block with division."""

    def __init__(self, n_layers=6, n_filters=64,
                 kernel_size=3, apply_sigmoid=True):
        super(DivideDespeckle, self).__init__()
        conv = BasicConv(in_channels=1, n_layers=n_layers,
                         n_filters=n_filters,
                         kernel_size=kernel_size)
        self.remove_noise = ResModel(
            conv,
            skip_connection=(lambda x, y:
                             x / (y + SAFE_DIV_EPSILON))
        )
        self.apply_sigmoid = apply_sigmoid

    def forward(self, x):
        clean_x = self.remove_noise(x)
        if self.apply_sigmoid:
            return torch.sigmoid(clean_x)
        return clean_x
```

Stain models

CycleGAN models are based on the implemetation in
`https://github.com/eriklindernoren/PyTorch-GAN`

```
import torch.nn as nn
```

```
import torch


# encoder block
class DownsamplingBlock(nn.Module):
    """Returns downsampling module of each generator block.

    conv + instance norm + relu
    """
    def __init__(self, in_features, out_features, normalize=True):
        super(DownsamplingBlock, self).__init__()
        layers = [nn.Conv2d(in_features, out_features, 3,
                            stride=2, padding=1)
                  ]
        if normalize:
            layers.append(nn.InstanceNorm2d(out_features))
        layers.append(nn.LeakyReLU(0.2, inplace=True))
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)


# decoder block
class UpsamplingBlock(nn.Module):
    """Returns UNet upsampling layers of each generator block.

    transposed conv + instance norm + relu
    """
    def __init__(self, in_features, out_features,
                 normalize=True):
        super(UpsamplingBlock, self).__init__()
        # multiply in_features by two because of
        # concatenated channels.
        layers = [nn.ConvTranspose2d(
                      in_features * 2,
                      out_features, 3,
                      stride=2, padding=1,
                      output_padding=1)
                  ]
        if normalize:
            layers.append(nn.InstanceNorm2d(out_features))
        layers.append(nn.LeakyReLU(0.2, inplace=True))
        self.model = nn.Sequential(*layers)

    def forward(self, x1, x2):
```

```
        x = torch.cat((x1, x2), dim=1)
        return self.model(x)


# ResNet block
class ResidualBlock(nn.Module):
    def __init__(self, in_features):
        super(ResidualBlock, self).__init__()

        conv_block = [nn.ReflectionPad2d(1),
                      nn.Conv2d(in_features, in_features, 3),
                      nn.InstanceNorm2d(in_features),
                      nn.ReLU(inplace=True),
                      nn.ReflectionPad2d(1),
                      nn.Conv2d(in_features, in_features, 3),
                      nn.InstanceNorm2d(in_features)]

        self.conv_block = nn.Sequential(*conv_block)

    def forward(self, x):
        return x + self.conv_block(x)


##################################
#         Generators
##################################

class AffineGenerator(nn.Module):
    """Affine transform generator implemented as a
    single layer 1x1 conv layer."""

    def __init__(self, input_nc, output_nc):
        super().__init__()
        self.model = nn.Conv2d(input_nc, output_nc, 1)

    def forward(self, x):
        x = self.model(x)
        return x


class GeneratorResNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=3,
                 res_blocks=9):
        super(GeneratorResNet, self).__init__()

        # Initial convolution block
```

```
        model = [nn.ReflectionPad2d(3),
                 nn.Conv2d(in_channels, 64, 7),
                 nn.InstanceNorm2d(64),
                 nn.ReLU(inplace=True)]

        # Downsampling
        in_features = 64
        out_features = in_features * 2
        for _ in range(2):
            model += [nn.Conv2d(in_features, out_features, 3,
                                stride=2, padding=1),
                      nn.InstanceNorm2d(out_features),
                      nn.ReLU(inplace=True)]
            in_features = out_features
            out_features = in_features * 2

        # Residual blocks
        for _ in range(res_blocks):
            model += [ResidualBlock(in_features)]

        # Upsampling
        out_features = in_features // 2
        for _ in range(2):
            model += [nn.ConvTranspose2d(in_features,
                                         out_features,
                                         3, stride=2,
                                         padding=1,
                                         output_padding=1),
                      nn.InstanceNorm2d(out_features),
                      nn.ReLU(inplace=True)]
            in_features = out_features
            out_features = in_features // 2

        # Output layer
        model += [nn.ReflectionPad2d(3),
                  nn.Conv2d(64, out_channels, 7),
                  nn.Tanh()]

        self.model = nn.Sequential(*model)

    def forward(self, x):
        return self.model(x)


class GeneratorUNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=3,
```

```
            num_down=2):
        super(GeneratorUNet, self).__init__()
        self.num_down = num_down
        self.down_activations = {}

        def get_activation(name):
            def hook(model, input, output):
                self.down_activations[name] = output
            return hook

        # Initial convolution block
        self.first = nn.Sequential(nn.ReflectionPad2d(3),
                                   nn.Conv2d(in_channels,64,7),
                                   nn.InstanceNorm2d(64),
                                   nn.ReLU(inplace=True))

        # Downsampling
        down_layers = []
        in_features = 64
        out_features = in_features * 2
        for i in range(self.num_down):
            down_layers.append(
                DownsamplingBlock(
                    in_features,
                    out_features)
                .register_forward_hook( get_activation(i)))
            in_features = out_features
            out_features = in_features * 2
        self.down_layers = nn.Sequential(*down_layers)

        # Middle
        self.middle = nn.Sequential(
            nn.Conv2d(in_features, in_features,
                      3),
            nn.InstanceNorm2d(in_features),
            nn.LeakyReLU(0.2, inplace=True)
        )

        # Upsampling
        up_layers = []
        out_features = in_features // 2
        for _ in range(self.num_down):
            up_layers.append(UpsamplingBlock(in_features,
                                             out_features))
            in_features = out_features
            out_features = in_features // 2
```

```
        self.up_layers = nn.Sequential(*up_layers)

        # Output layer
        self.last = nn.Sequential(nn.ReflectionPad2d(3),
                                  nn.Conv2d(64,out_channels,7),
                                  nn.Tanh())

    def forward(self, x):
        out_first = self.first(x)
        out_encoder = self.down_layers(out_first)
        out_middle = self.middle(out_encoder)
        for i, decoder_layer in enumerate(self.up_layers):
            if i == 0:
                out = decoder_layer(
                    self.down_activations[self.num_down-1-i],
                    out_middle)
            else:
                out = decoder_layer(
                    self.down_activations[self.num_down-1-i],
                    out)

        return self.last(out)


##################################
#         Discriminator
##################################

class Discriminator(nn.Module):
    def __init__(self, in_channels=3, discriminator_blocks=4):
        super(Discriminator, self).__init__()

        def discriminator_block(in_filters, out_filters,
                                normalize=True):
            """Returns downsampling layers of each
            discriminator block."""
            layers = [nn.Conv2d(in_filters,
                                out_filters,
                                4,
                                stride=2,
                                padding=1)]
            if normalize:
                layers.append(nn.InstanceNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers
```

```
        n_filters = 64
        blocks = discriminator_block(in_channels, n_filters,
                                     normalize=False)
        for _ in range(discriminator_blocks - 1):
            blocks += discriminator_block(n_filters,
                                          n_filters * 2)
            n_filters *= 2

        self.model = nn.Sequential(
            *blocks,
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.Conv2d(n_filters, 1, 4, padding=1)
        )

    def forward(self, img):
        return self.model(img)
```