Department of Mathematics
Chair of Mathematical Modeling of Biological Systems

Predicting transcription rate from multiplexed protein maps using deep learning

Master's Thesis by Andres Becker

Examiner:	Prof. Dr. Fabian J. Theis
Advisor:	Dr. Hannah Spitzer
Submission Date:	April 14, 2021

Declaration

I hereby declare that this thesis is my own work and that no other sources have been used except those clearly indicated and referenced.

Andres Alberto Becker Sanabria, München, 15.05.2021

Abstract

Here we give a short summary of the project or thesis of length at most a quarter of a page.

- What is the objective of this work
- How did we address it
- What did we obtain

Hannah's recommendations:

- 1 sentence to set the background: what is currently done
- 1 sentence to capture the issue: why is this an issue/why can we do better?
- 1 sentence to describe your solution: We propose to...
- 1 sentence to give an overview of your results
- 1 sentence to state the impact: what will this allow people to do

Acknowledgments

To my father, my partner in my wildest adventures, best friend and who taught me what are the important things in life. He may never read this, but let the world known he is a loved and admired man.

Contents

Chapter 1

Introduction

A small overview. Explain how CNN can be used to predict Transcription Rate and how we can use interpretability methods to learn from this models.

Hannah's recommendations about intro chapter:

- Problem statement and challenges
- Main objectives / research questions
- List contributions
- last section of intro is important; give summary of content + key impact

Hannah's general recommendations:

- Content: You do not need to include all models you trained and details of all the bugs you fixed, but the thesis should contain a consistent story (detailled in the contributions in the introduction) where every result you show adds something to the story. For every experiment/figure ask yourself: is this necessary to answer the research question? If the answer is no, don't include it
- Assume that some people will only read the introduction and summary, and look at the figures -> the main points should be understandable from this information alone
- The thesis should be written at a level where a masters student from your course of study can understand it. All concepts that are not know at this level, need to be introduced
- Prefer descriptive section and paragraph headings that ideally tell the reader what this section is about rather than the method that was used

Hannah's recommendations about figures:

- Give references for every figure you use

- Describe all figures in the main text, but also make sure that each figure has a descriptive caption.

- Captions should help describe what is being shown, i.e. what do the colors stand for etc. They should help the reader to understand the flow of the figure. They shouldn't contain too much methods stuff (i.e. how this was done -> belongs to the methods) neither should they contain results stuff (how do we interpret these results? what do they mean? -> belongs to the results). The figure and caption should stand alone and be interpretable without the rest of the text. Ideally you should be able to redraw the figure from the caption and recreate the caption from the figure (but that is a very high bar).

1.1 Motivation

Understanding how RNA concentration in eukaryotes cells is regulated is very important to understand gene expression. Measuring the amount of RNA inside a cell, may not be enough to fully describe cellular function. Accordingly to Buxbaum et al. [BHS14] and Korolchuk et al. [Kor+11], cellular function can heavily depends on the specific intracellular location and interaction with other molecules and intracellular structures. Therefore, the use of models capable of focusing on the spatial information contained in cell images, can potentially help us to understand cell expression.

1.2 Goal of this Thesis

The objective of this work is to predict the RNA concentration within the null of a cell using Convolutional Neural Networks (CNNs) and multiplexed images of the cell nucleus. These multiplexed images capture the distribution and location of several molecules and proteins, as well as the shape and location of some organelles within a cell nucleus.

Nevertheless, this work goes beyond just trying to predict RNA concentration. Through interpretation methods, we can observe the places in the cell nucleus image that were most relevant to the model when making its prediction (score maps). To do this, we implemented preprocessing and data augmentation techniques aimed to improve the model's training performance and prevent overfitting, but most importantly, to remove non-relevant information from images. This encouraged the model to focus mainly on spatial information and allowed us to obtain cleaner score maps, which can potentially help to understand cellular expression better.

Finally, we test the veracity of the information in the score maps, by confronting them with a validation method.

1.3 Literature review

In recent years neural networks...

However, once we have successfully trained an artificial neural network to make predictions, one may also want to understand how the predictions was made. Several approaches have being used to tackle this. In this work we focus in the ones that highlight features in an input that are most responsible for the model's output. We use a composition of two explainable AI techniques, Integrated Gradient (IG) [STY17] to highlight relevant input components (saliency map) and VarGrad (VG) [Ade+20] to enhance the empirical quality the saliency map.

Hannah's recommendations:

- introduce most relevant previous work
- Set it in context: how does this relate to your work?

Chapter 2

Basics

In this chapter we will... A small overview. Explain how CNN can be used to predict Transcription Rate and how we can use interpretability methods to learn from this models.

Hannah's recommendations:

- explain biological background
- introduce all methods that are used later on
- Experts might skip reading this chapter so don't include any
- information on your dataset, specific evaluation etc here

2.1 Biology Background

Cells are considered the smallest unit of life. There are two types of cells, *prokaryotic* and *eukaryotic*. The main difference between these, is that prokaryotic cells do not contain nucleus and that that prokaryotes are considered single-celled organisms, while eukaryotes organisms can be either single-celled or multicellular. For multicellular organisms, like plants or mammals, eukaryotic cells are the *building-blocks* of life. This work focuses on a process of eukaryotic cells. Therefore, in the subsequent when we refer to cells, we will be referring to eukaryotic cells only.

Multicellular organisms (like us) have different cell types, where each one of them can have many or an specific function. For instance, red blood cells are responsible for carrying the oxygen in the body. In order to carry as much oxygen as possible they lack a nucleus, and therefore they are unable to undergo *mitosis*[1].

However, there are also cells aimed to produce (*synthesize*) certain substances that regulate process in our body. For instance, *Alpha cells* are pancreatic cells responsible for synthesizing the *glucagon* hormone, which elevates the glucose levels in the blood [Ker99]. The process in which cells produce this substances is called *cellular expression*

[1] Mitosis is the process through which eukaryotic cells reproduce themselves and give rise to new organisms.

or *gene expression*. The reason why this process is also called gene expression, is because the instructions to synthesize every substance (or any functional product, like hormones or proteins) are encoded in a specific gene[2].

There are two key steps involved in gene expression, *transcription* and *translation*. Roughly speaking, transcription is the process in which the instructions to synthesize a product (like proteins) are copied from a gene in the DNA, to a single strand molecule called messenger RNA (mRNA). On the other hand, Translation is the process in which the instructions in the mRNA are interpreted to produce a functional product. Figure 2.1 shows a simple representation of this process.

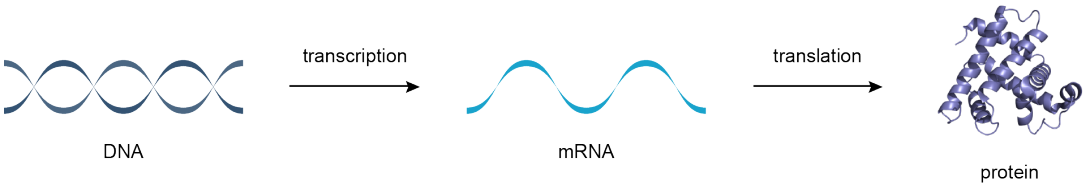


Figure 2.1: Simple representation of the gene expression process [BJ].

The transcription process happens inside the cell nucleus, while translation happens in the *ribosome* (outside the nucleus). The reason why transcription is necessary, is because the instructions needed to build a product are encoded in the DNA, which is inside the nucleus. Since DNA is too big to pass the membrane that covers the nucleus (nuclear envelop) to travel to the ribosome (which is the organelle in charge of building the product), the necessary instructions in the DNA are copied into a smaller strand (mRNA), which is now able to escape the nucleus and travels to the ribosome to start the translation process. Figure 2.2 shows a diagram of an eukaryotic cell and some of its parts. This work focuses on the transcription process and the factors that seed up or slow down this process.

2.1.1 Transcription Process

Transcription is the process in which the instructions encoded in a gene are copied from the DNA inside the nucleus, to produce a RNA transcript called messenger RNA. The Transcription process has two broad 2 main steps; 1) The process in which the gene is copied from the DNA into a pre-processed version of the messenger RNA (mRNA) called pre-messenger RNA (pre-mRNA); 2) The RNA Splicing, which is the process where the pre-mRNA is transformed into a mature mRNA strand. These two processes happen inside the cell nucleus.

[2] A *gene* is defined as a region of the *DNA* that encodes a function. DNA is contained in *chromosomes*, which are long DNA strands containing many genes.

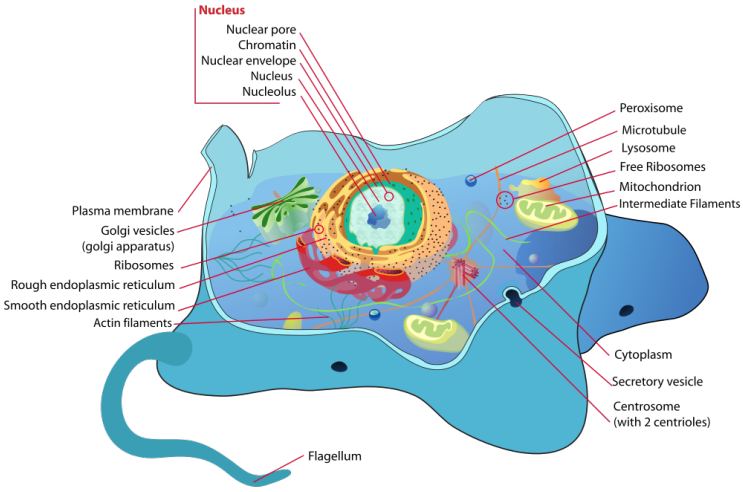


Figure 2.2: Animal eukaryotic cell diagram [Rui].

Step 1, Pre-messenger RNA synthesis

The pre-mRNA creation has 3 main processes[JD+13] and is illustrated in figure 2.3:

1. **Initiation**. This step initiate the transcription process. It happen when an *enzyme*[3] of RNA polymerase binds to the a region of the target gene called *the promoter*. This indicate the DNA to unwind, so the RNA polymerase can read the DNA bases in one of its strands and create a molecule of pre-mRNA.

2. **Elongation**. Elongation is the process in which *nucleotides*[4] are added to the pre-mRNA strand. The RNA polymerase enzyme read the unwound DNA strand and synthesize the pre-mRNA molecule.

3. **Termination**. Termination ends the pre-mRNA synthesis. It happens when the RNA polymerase enzyme identifies a termination sequence in the gene, and detaches from the unwound DNA.

[3] An enzyme is a proteins that act as biological catalysts to accelerate chemical reactions.

[4] Nucleotides are the building block of nucleic acids. A nucleotide consists of a sugar molecule bound to a phosphate group and a nitrogen-containing base. RNA and DNA are polymers made of long chains of nucleotides.

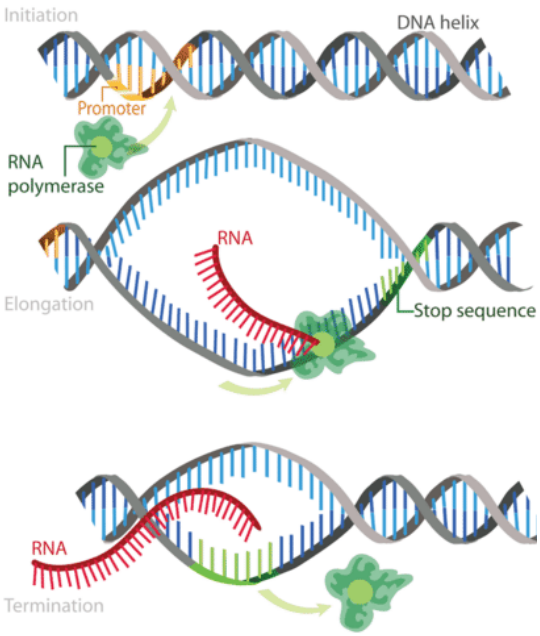


Figure 2.3: The three main steps of the pre-mRNA synthesis: initiation, elongation, and termination [Vil].

Step 2, Pre-messenger RNA splicing

In this process the pre-mRNA is transformed into a mature mRNA strand by removing non-relevant (non-coding) sections of it. During Splicing, *introns*[5] are removed and *exons*[6] are joined together [Ber+15]. Besides this, a cap and a tail are added to the spliced pre-mRNA strand to turn it into a mature mRNA. Figure 2.4 illustrates the pre-mRNA splicing process.

Transcription Rate

NOTE: Solo define aqui el transcription rate en general, la parte de Methodology Dataset

Here I should state what is transcription rate (TR). In this part I will not mention that we use the average of the channel `00_EU`, that will be stated in the methodology part. However, here I should state a definition of TR that agrees with what we have in the dataset. Here are some ideas of what TR is. Since we have images of cell nucleus,

[5]Introns are nucleotide sequences within a gene that are non-coding regions of an RNA transcript, and that are removed by the splicing process before translation.

[6]Exons are coding sections of an RNA transcript that can be translated into proteins.

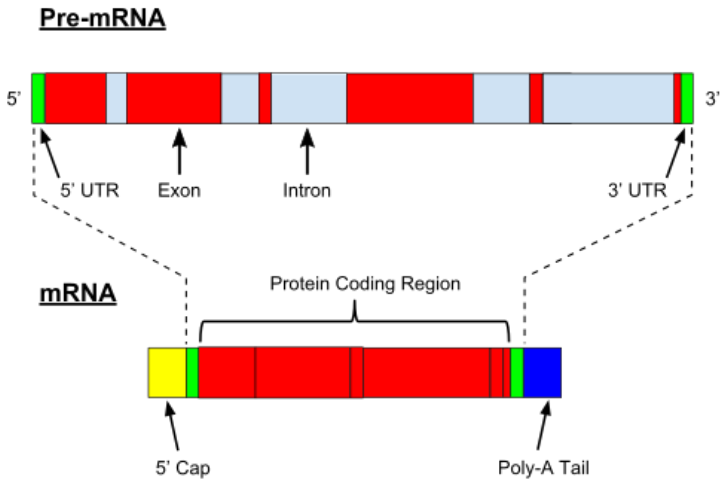


Figure 2.4: Pre-messenger RNA splicing process. A pre-mRNA strand (top) is turned into a mature mRNA strand (bottom) [Wik21].

this means that we have both pre-mRNA and mature mRNA molecules[7]. So, what is `00_EU` measuring? both molecules or only one? As far as I understood, `00_EU` captures the RNA molecules that were produced in the last 30 minutes. Therefore, for us TR would be the average concentration of mRNA per unit of time (in this case 30 mins). However if we measure the concentration of pre or mature mRNA in the nucleus, then in the TR definition we should consider either (or both) the rate that pre turns into mature (but if we measure both then we mesure the total and this is not necessary) or the rate at which mature mRNA escapes the nucleus. FiXme Note!

Therefore, accordingly to [PO+13] we can define TR as the number of mRNA molecules being synthesized per unit time minus a degradation factor

$$\frac{d[mRNA]}{dt} = SR - k_d[mRNA] \tag{2.1}$$

where $[mRNA]$ is the mature mRNA concentration in the cell nucleus, RN the mRNA synthesis rate and k_d the degradation rate (or in our case, the rate at which mature mRNA escapes the nucleus). FiXme Note!

[7] Transcription rate encompasses two related, yet different, concepts: the nascent transcription rate, which measures the in situ mRNA production by RNA polymerase, and the rate of synthesis of mature mRNA, which measures the contribution of transcription to the mRNA concentration [PO+13].

2.2 Machine Learning

Artificial Neural Networks (ANNs) are universal approximators widely used in the field of Machine Learning (ML) and an important part of this work. This is a very broad subject and there are entire books that cover this in detail, like [GBC16] or [Bis06]. However, in this section we will give a small introduction to ANNs and Convolutional Neural Networks (CNNs), which are a type of ANN that were specifically designed to deal with data in the form of images.

Before defining what exactly is a ANN, lest first recall the definition of machine learning. We refer as ML to the group of algorithms that automatically improve (learn) through experience. Among this algorithms, we could say that there are three main classes (which depend on the kind of experience we provide):

- **Supervised Learning**: The experience is given in the form of input and output examples, and the goal is to learn a general rule that maps inputs to outputs.
- **Unsupervised Learning**: The experience is given in the form of data (no outputs provided) and the goal is to discover hidden patterns in data.
- **Reinforcement Learning**: No experience (data) is given, instead a dynamic *environment* is provided and an *agent* must learn how to interact with it in order to achieve a goal.

An ANN can be used in any of the 3 kinds of learning algorithms listed above.

However, recall that the objective of this work is to approximate a function (in this case a CNN), such that when it is fed with images of a cell nucleus (input data), it predicts the corresponding transcription rate (TR) (output data) Therefore, we are dealing with a *supervised learning* task.

Before explaining what a CNN is, let us first introduce and explain ANN in general.

2.2.1 Artificial Neural Networks

Roughly speaking, an Artificial Neural Network (ANN) is a non-linear function $f : \mathbb{R}^D \to \mathbb{R}^L$, that maps an input $\boldsymbol{x} \in \mathbb{R}^D$ with an output $\boldsymbol{y} \in \mathbb{R}^L$. Of course, to consider f as a ANN, f must have a specific form that will be address later. However, for the sake of this explanation, let us start by defining a simple function as follow

$$\begin{aligned} f(\boldsymbol{x}, \boldsymbol{w}) &:= h\left(w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\boldsymbol{x})\right) \\ &= h(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x})) \\ &:= h(z) \end{aligned} \tag{2.2}$$

where $\boldsymbol{\phi} : \mathbb{R}^{D+1} \to \mathbb{R}^M$ is an element-wise function, with $\phi_0 := 1$, know as *basis function*, $h : \mathbb{R} \to \mathbb{R}$ is a function know as *activation function* and $\boldsymbol{w} \in \mathbb{R}^M$ is the parameter vector. The parameters w_j, with $j \in \{1, \dots M-1\}$ are known as *weights*, while the parameter w_0 is know as *bias*.

Then, an ANN is composition of functions of the same form as 2.2, with non-linear *activation functions*, and where the basis functions are also of the same form as 2.2 [Bis06]

$$F(\boldsymbol{x}, \boldsymbol{W}) := h_K(\boldsymbol{w}_K^T h_{K-1}(\boldsymbol{w}_{K-1}^T \dots h_0(\boldsymbol{w}_0^T \boldsymbol{x}) \dots)) \tag{2.3}$$

The subscript in the parameter vectors $\boldsymbol{w}_k$ and the activation functions h_k, with $k \in \{0, \dots, K\}$, of 2.3 represents the depth of the layers. Note that unlike the other layers, the base function of the *input layer* (k = 0) is the identity function. Furthermore, the activation function of the *output layer* h_K does not necessarily have to be non-linear. Instead, it is chosen based on the type of function we want to approximate. In our case, since we have a regression problem (predicting transcription rate (TR)), h_K is chosen as the identity function.

There are different non-linear activation functions that can be chosen for the hidden units. However, all the models showed in this work use the Rectified Linear Unit (ReLU)

$$ReLU := max\{0, x\} \tag{2.4}$$

Figure 2.5 shows the ReLU activation function.

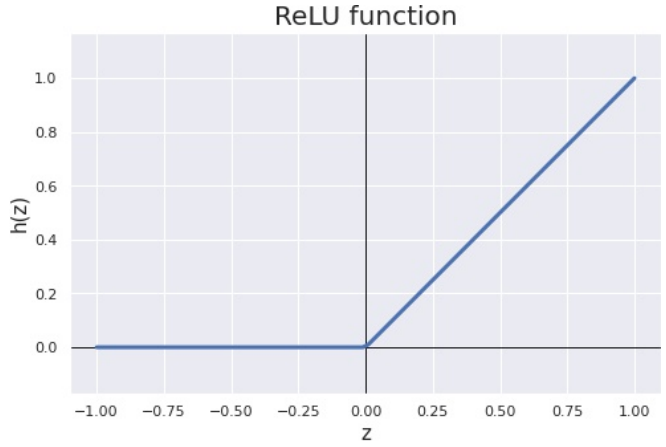


Figure 2.5: ReLU activation function.

Figure 2.6 shows a graphical representation of a ANN. The circles represent the activation function applied to what is inside it. Black colored circles represent the

identity function, red colored circles the non-linear activation function for the hidden layers, while green any function for the output layer that suits the problem we want to solve. Note that values inside the circles of the hidden and output layers z_i^k, for $k \in \{0, \dots, K\}$ and i representing one of the units of the k layer, are the output of a function of the same form as 2.2. The lines connecting the circles represent the weights and biases corresponding to each layer $\boldsymbol{W}_k$, for $k \in \{0, \dots, K\}$. The circles in the *hidden layers* are known as *hidden units*.

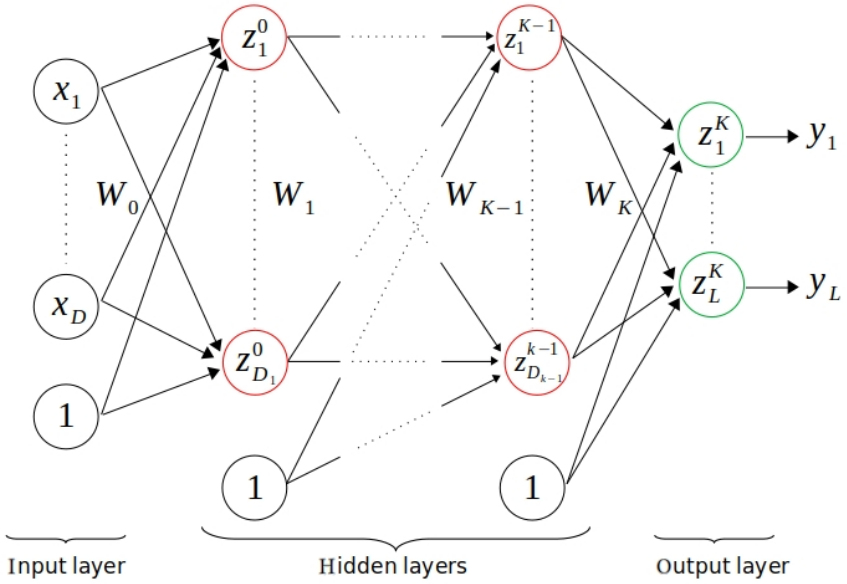


Figure 2.6: Graphical representation of an ANN. The color of the circles represents the type of activation function. Black means the identity, red a non-linear function for the hidden layers and green any function for the output layer.

Strictly speaking, equation 2.3 and figure 2.6 represent a *fully connected feedforward neural network*. However, in this work we will refer to it just as ANN, which in some literature is also known as Multilayer Perceptron (MLP). Also, hidden layers are also known as *Dense layers*.

Update rule

Sow far we have introduced the general form an ANN must have. Moreover, equation 2.3 shows that an ANN is simply a non-linear function controlled by a set of adjustable parameters $\boldsymbol{W}$. Therefor the question is, how can we approximate this parameters?

Recall that we are dealing with a supervised learning problem, which means that we can use both the input data (images of cell nucleus, $\boldsymbol{X}$) and the output data (the TRs,

$\boldsymbol{Y}$) to approximate $\boldsymbol{W}$. Therefore, we can fed the ANN with $\boldsymbol{X}$, and then measure its performance by comparing its output $\hat{\boldsymbol{Y}}$ against the true values $\boldsymbol{Y}$.

This comparison is made by means of a *loss function* $\mathcal{L}$ that must be chosen beforehand. The choice of $\mathcal{L}$ depends mainly on the type of problem you are solving (regression, classification, etc.). However, even for each type, there are many different options. For now, let us just say that $\mathcal{L}$ should return high values when $\hat{\boldsymbol{Y}}$ is far from the true values $\boldsymbol{Y}$, and low when they are close.

Then, we can fit the values of $\boldsymbol{W}$, by minimizing the loss function $\mathcal{L}$ each time the model is fed with an input value $\boldsymbol{x}$. Since the gradient of $\mathcal{L}$ with respect to $\boldsymbol{W}$ (i.e., $\nabla_{\boldsymbol{W}}\mathcal{L}$) returns the direction in which the loss function grows the fastest, then we choose $-\nabla_{\boldsymbol{W}}\mathcal{L}$ as the direction of our update rule

$$\boldsymbol{W}_{new} = \boldsymbol{W}_{old} - \alpha\nabla_{\boldsymbol{W}}\mathcal{L}(\boldsymbol{W}_{old}) \tag{2.5}$$

where $\alpha \in \mathbb{R}^+$ (known as *learning rate*) controls how much we move in the direction of $-\nabla_{\boldsymbol{W}}\mathcal{L}(\boldsymbol{W}_{old})$ on every step.

The iterative method in which 2.5 is applied over elements of $\boldsymbol{X}$ to optimize $\boldsymbol{W}$ is known as Gradient Descent (GD) [Bis06]. However, in practice 2.5 is not applied for a single element of $\boldsymbol{X}$ every time, but to a random subset of $\boldsymbol{X}$ (known as a *Batch*) instead. The number of elements in batch is fixed over all the iteration (training), and is an hyperparameter known as *Batch Size* bs[8]. As a rule of thumb, bs should be no less than 30 (for the selected sample to be representative of $\boldsymbol{X}$). In practice bs is usually chosen as a power of 2. This process is known as Stochastic Gradient Descent (SGD) and computationally is less expensive than GD.

However, GD (SGD) has a downside, the choice of its hyperparameter α (learning rate). In practice, it has been shown that the correct choice of α is essential to train an ANN successfully. Therefore, other algorithms (*optimizers*) have been proposed to mitigate this problem. The revision of these optimizers is out of the scope to this work. However, all of them follow the same idea proposed by GD. For example, instead of having a fixed learning rate α as in GD, the Adaptive Moment Estimation (Adam) optimizer adapts its learning rate dynamically during training depending on the mean and variance of the loss function [KB14].

Back propagation

Nevertheless, there is still one question that needs to be answered, which is how to efficiently calculate the derivative of the loss function ($\nabla_{\boldsymbol{W}}\mathcal{L}$) with respect to all the parameters of the ANN. The answer to this is through an algorithm called *backpropagation*, which is performed during the *training process*. Again, there is a lot

[8] Normally the training data is separated in disjoint batches, which means that it could happen that last batch to be smaller than the selected bs.

of literature that explains this in depth(for instance [GBC16] or [Bis06]). Therefor, here we will just provide the intuition behind it.

Recall that $\mathcal{L}$ is a function of the true values y and $\hat{y}$ i.e., $\mathcal{L}(y, \hat{y})$. Also from equation 2.3 and figure 2.6 note that

$$\begin{aligned} y &:= F(\boldsymbol{x}, \boldsymbol{W}) \\ &= h_K(\boldsymbol{z}^K) \\ &= h_K(\boldsymbol{W}_K^T h_{K-1}(\boldsymbol{z}^{K-1})) \end{aligned} \tag{2.6}$$

and therefore

$$\begin{aligned} \nabla_{\boldsymbol{W}_K} \mathcal{L} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_K} \\ &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{z}^{\boldsymbol{K}}} \frac{\partial \boldsymbol{z}^{\boldsymbol{K}}}{\partial \boldsymbol{W}_K} \end{aligned} \tag{2.7}$$

which is just the product of the derivative of the loss function w.r.t. $\hat{y}$ (i.e., $\frac{\partial \mathcal{L}}{\partial \hat{y}}$), the derivative of the activation function of the output layer w.r.t the argument of the last layer (i.e., $\frac{\partial \hat{y}}{\partial \boldsymbol{z}^{\boldsymbol{K}}}$) and the output of the layer $K-1$ (i.e., $\frac{\partial \boldsymbol{z}^{\boldsymbol{K}}}{\partial \boldsymbol{W}_K} = h_{K-1}(\boldsymbol{z}^{\boldsymbol{K-1}})$).

Note that we can easily compute the gradient of $\mathcal{L}$ w.r.t deeper parameters $\boldsymbol{W}_k$ (for $k \in \{0, \ldots, K-1\}$), just by extending 2.6 and 2.7.

This shows how by means of the *chain rule*[9], the backpropagation algorithm can compute the gradient of the loss function w.r.t. a specific parameter, just by multiplying the derivative of the loss function, the derivative of the activation functions and some values computed during the evaluation of the ANN.

Model development

The properties of ANNs have been studied extensively before ([Cyb89], [HSW89], [Fun89]) and established in the *Universal approximation theorem*

Theorem 2.1 (Universal approximation theorem)
An MLP with a linear output layer and one hidden layer can approximate any continuous function defined over a closed and bounded subset of $\mathbb{R}^D$, under mild assumptions on the activation function (squashing activation function) and given the number of hidden units is large enough.

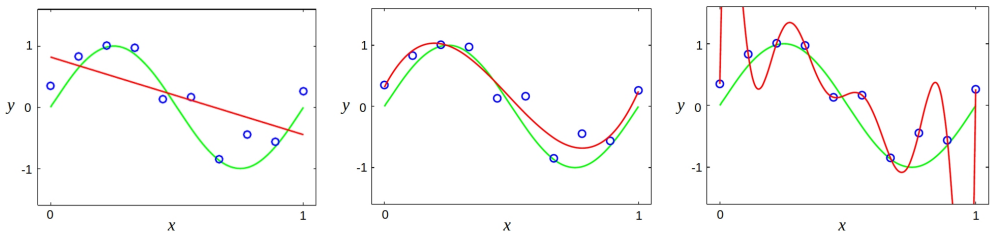
[9] $(f \circ g)' = (f \circ g) \cdot g'$, or equivalently $h'(x) = f'(g(x))g'(x)$, for $h(x) := f(g(x))$.

For this reason ANN are known as *universal approximators*, since they are able to approximate any continuous function on a compact[10] input domain with an arbitrary accuracy [Bis06].

These means that, as long as a ANN has a sufficiently large number of hidden units, the loss function can be reduced as much as desired. However, this nice property can also lead to an unwanted one known as *overfitting*. Intuitively this means that the ANN *memorize* the data used to train it (low error/bias), and therefore it is not able to perform (or *generalize*) well when it is fed with new data (high error/bias and variance). This happens mainly when the ANN is optimized/fed too many times with the same data.

On the other hand, *underfitting* means that the ANN performs poorly on both new data and data used to train the network (high bias and low variance). This usually happens when the training time is insufficient or the ANN is not complex enough (too few hidden units and/or layers).

Figure 2.7 shows synthetic data (blue circles), generated from a sine function (green line) and random noise sampled from a normal distribution. The red line in figure 2.7a represents a fitted model with high bias and low variance (underfitting), while in figure 2.7c a model with low bias and high variance (overfitting). The red line in figure 2.7b, represents a model with low bias and variance (good fit and good generalization).



(a) Underfitted model. **(b)** Model with good fit and generalization. **(c)** Overfitted model.

Figure 2.7: Representation of a model (red line) with underfitting a), good fit b) and overffiting c), trained over synthetic data (blue small circles). The synthetic data was generating by adding random noise to a sine function (green line) on the interval $[0, 1]$. Image source [Bis06].

In practice, we seek to fit models that has low bias and low variance (i.e., good accuracy and good generalization). Therefore, to prevent overfitting we split the data into 3 different sets; *training*, *validation* and *test*, and train the model using only the first set. Then, during model training, we measure how well the model is generalizing

[10] A set A in a metric space is said to be *compact* if it is close (i.e., it contain all its limit points) and bounded (i.e., all its points lie within some fixed distance of each other) [BS00].

by comparing the value of the loss function when it is evaluated in the training and validation set [11]. During the model development, the *test* set is never evaluated and is only used at the end, to report the model performance. This methodology is shown in figure 2.9

Figure 2.8 shows this *bias–variance tradeoff* between training and validation set. In practice, multiple versions of the model are saved during training and then the one with the lowest validation error is chosen (red dot on figure 2.8).

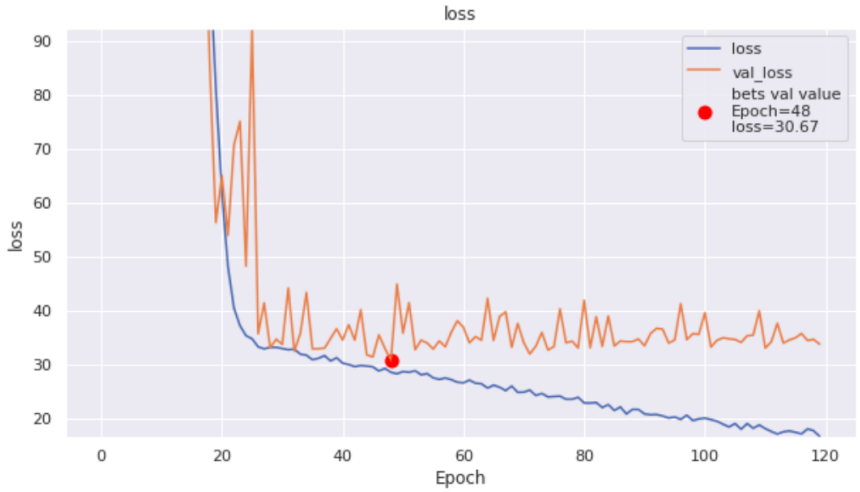


Figure 2.8: Bias–variance tradeoff. In orange (respectively blue) the loss function curve when it is evaluated in the validation (respectively training) set. The red dot shows the lowest loss for the validation set.

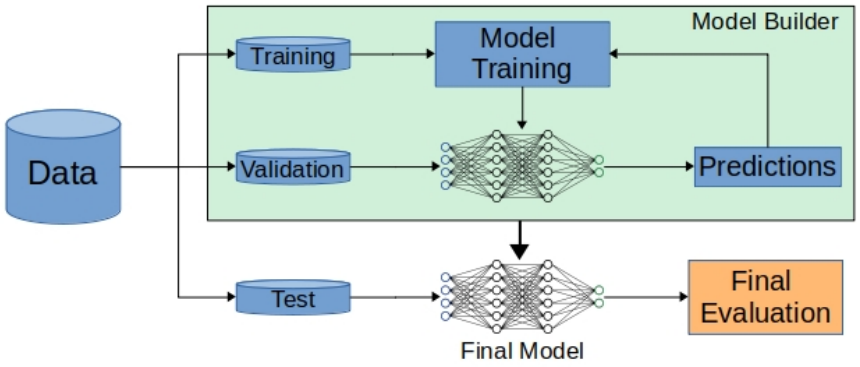


Figure 2.9: Model development methodology.

[11] This is usually known as the *bias–variance tradeoff*.

The methodology shown in figure 2.9 is also used to optimize the hyperparameters of the model, like the number hidden units/layers or the activation function of the hidden layers.

Batch Normalization

However, overfitting is not the only problem we may encounter when training an ANN. Training ANN with several layers can be complicates, since the distribution of the data can change from layer to layer. This means that the input and output distribution of a layer will not necessarily be the same. It has been empirically proven that this can affect the training performance, since it require the use of lower learning rates [IS15]. This can also lead to *saturation*[12] of the activation functions, so a more careful initialization of the ANN parameters is required. To address this problem Ioffe et al. [IS15] proposed to normalize the layer inputs.

Roughly speaking, batch normalization consist of two main steps; 1) the standardization of the layer input and 2) the normalization of the standardized data. For the first step the layer input is standardized using parameters extracted from the *batch*

$$\boldsymbol{z}'_k := \frac{\boldsymbol{z}_k - \boldsymbol{\mu}_k}{\sqrt{\boldsymbol{\sigma}_k^2 - \epsilon}} \tag{2.8}$$

with

$$\begin{aligned} \boldsymbol{\mu}_k &= \frac{1}{M}\sum_{m=1}^{M} \boldsymbol{z}_k \\ \boldsymbol{\sigma}_k^2 &= \frac{1}{M}\sum_{m=1}^{M} (\boldsymbol{z}_k - \boldsymbol{\mu}_k)^2 \end{aligned} \tag{2.9}$$

where M is the *Batch* size and k, with $k \in \{0 \dots K\}$, denotes the layer.

Note that for each layer k we have different normalization parameters $\boldsymbol{\mu}_k$ and $\boldsymbol{\sigma}_k$. Moreover, this normalization parameters are vectors of the same shape as the layer size (i.e., one pair of normalization parameters per unit/neuron).

The second step in batch normalization consist on normalizing the standardized data $\boldsymbol{z}'_k$ using parameters $\boldsymbol{\gamma}_k$ and $\boldsymbol{\beta}_k$ learned during training

$$\widetilde{\boldsymbol{z}}_k := \boldsymbol{\gamma}_k \odot \boldsymbol{z}'_k + \boldsymbol{\beta}_k \tag{2.10}$$

where $\odot$ denotes *element-wise* multiplication. At the beginning of the training $\boldsymbol{\gamma}_k = 1$ and $\boldsymbol{\beta}_k = 0$ are used for all the layers and units.

[12] *Saturation* is a commonly used term to refer to the situation when the evaluation of a "squashing" function returns values close to some of its horizontal asymptotes most of the time. Remember that these "squash" functions (like *Sigmoid* or *tanh*) compress the real line $(-\inf, \inf)$ into an interval of finite length (a, b).

During training, the normalization parameters of each epoch are stored, so the average ($\bar{\gamma}_k$ and $\bar{\beta}_k$) can be used during evaluation (when the model is not training).

Residual Block V2

As already mentioned, the *Universal approximation theorem* guarantees that the training error can be reduced by adding more layer to an ANN. However, in practice it is not that simple. As we add layers to an ANN, the training becomes more unstable and difficult as we can face vanishing or exploding gradients (when the value of the gradients become very close to 0 or inf respectively during back propagation). To overcome this problem, He et al. ([He+15] and [He+16]) proposed the *residual blocks*, which have been empirically shown to make deep ANN training more stable. The core idea of residual blocks is to reformulate the layers as *learning residual functions* with reference to the layer inputs, by adding an *identity connection*. Then, if a layer is not longer beneficial to the ANN (e.g. in case of gradient vanishing), the ANN can just "skip" it. Figure 2.10 shows a diagram of the second version of a residual block [He+16].

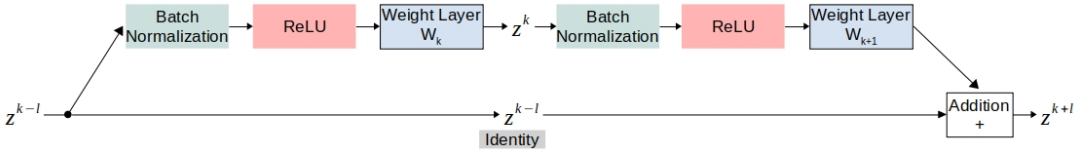


Figure 2.10: Residual block V2.

2.2.2 Convolutional Neural Networks

So far we have explained how Artificial Neural Networks (ANNs) works assuming that we feed them with vectors of fixed length. Even though we could take a multichannel image and transform it into a vector, in practice this would be computationally very expensive. For instance, assuming that we have a 3 channel image of size 224 by 224, this would result into an input vector of length $3 \cdot 224 \cdot 224 = 150'528$. Then, if the first layer of our network has 100 units, this would mean more than 15 millions of parameters only for the first layer. Furthermore, the transformation of our image into a vector would mean a loss of spatial information. This means that the ANN would not be able to capture or use the spatial relationship between pixels and shapes within the image.

A Convolutional Neural Network (CNN) is a type of ANN widely used to analyze data in the form of images. The intuition behind a CNN is that instead of just looking at an image and trying to predict the target value directly, first learn some *features* within the image, and then make the predict base on this features. To achieve this, CNNs mainly use *convolution* and *pooling* layers.

Convolution layer

The only difference a

A convolution layer is very similar to a regular layer described in section 2.2.1. Basically, they only differ in the way the layer input is multiplied by the the layer weights. Recall that in a regular layer, the input of a unit is the dot product between the layer input and its corresponding weight vector (i.e., $z = \boldsymbol{w}^T \boldsymbol{x}$). This means that for each element in the input vector $\boldsymbol{x}$, there is a corresponding element in the weight vector $\boldsymbol{w}$. However, for a convolution layer this is not the case. Convolution layers are based on the shared-weight architecture of the convolution *kernels* or *filters* that slide along the input and returns a translation known as *feature maps* [Zha+88]. This means that the *kernels* weights will be used for multiple elements of the layer input. Figure 2.11 shows the convolution process with a 2 by 2 kernel over a RGB image (3 channels) of size 4 by 4. Each entrance of the returned feature map z_i is the dot product between the kernel weights $\boldsymbol{w}$ and the $\boldsymbol{x}_i - th$ chunk of the image.

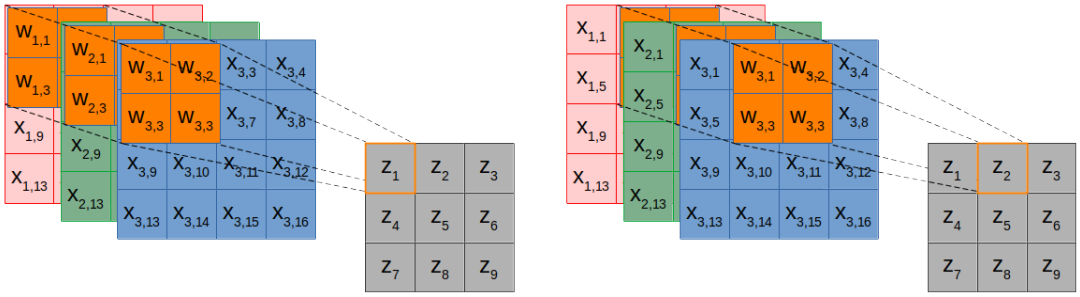


Figure 2.11: Convolution process steps. In red, green and blue the input image, in orange the convolution kernel (size 2 by 2 and stride of 1) and in gray the convolution output (feature map).

Mathematically this looks as follow

$$z_i = \boldsymbol{w}^T \boldsymbol{x}_i + b \tag{2.11}$$

where $\boldsymbol{w} \in \mathbb{R}^{2\times2\times3}$, $\boldsymbol{x_i} \in \mathbb{R}^{2\times2\times3}$ and $b \in \mathbb{R}$ is the bias (not shown in the images).

Like the kernel size, the number of pixels we shift the kernel each time along side the input (*Stride*) is also a hyperparameter of convolution layers. IN figure 2.11, the stride size is 1.

Figure 2.11 also shows that size (width and height) of the returned feature map is smaller than the input image. If we want to keep the input and output size the same (*Same convolution*), then we must add zeros at the edges of the input features (zero-padding). This is shown on figure 2.12.

So far we have seen that a convolution projects a multi-channel input feature (image)

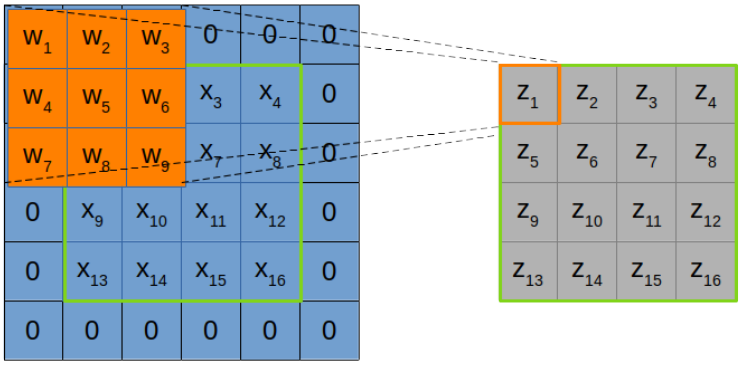


Figure 2.12: Convolution with padding. In blue a single-channel input features, in orange the convolution kernel (size 3 by 3 and stride of 1) and in gray the convolution output (feature map).

into a single-channel feature map. Therefore, if we want our output feature map to have n channels, then our convolution must have n different kernels.

Normally, a non-linear activation function is applied to the output of convolution layers (and normally also after batch normalization) to enable the CNN to learn non-linear relations.

Pooling layer

Unlike convolution layers, the goal of Pooling layers is to reduce the feature image (height and width, but not depth) rather than learn features. However, Pooling layers work in a similar way to convolution layers in the way that they also slide a kernel along the input. However, in this case the kernel works independently on each feature map (that is, each channel) and has no weights to learn. This means that the pooling layers maintain the same number of input and output channels. There are several ways to do this downsampling, but the most common are Max Polling and Average Polling. As the name suggests, Average pooling shrinks the feature image by averaging sections of it, while Max pooling takes the maximum value. Figure 2.13 shows an example of a max and average pooling layer on a single-channel feature image using a 2 by 2 kernel and a stride of 2.

Normally, Pooling layers are applied over the output of the activation functions.

Global Average Pooling layer

As we mentioned at the beginning of this section, the idea of a CNN is to first learn the features within the input images and then make a prediction based on these features. To do this, the *Global Average Pooling layer* transforms the channels of the last feature

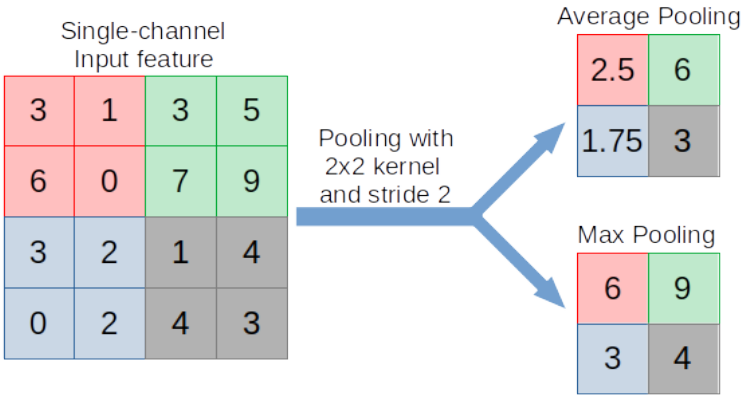


Figure 2.13: Max and average pooling with a 2 by 2 kernel and stride 2. The color denotes the kernel position.

map into a vector (by averaging each of its channels), so that this can be used as input in a regular ANN to make the final prediction. Figure 2.14 shows an example of this, when it is applied into a feature map with 7 channels.

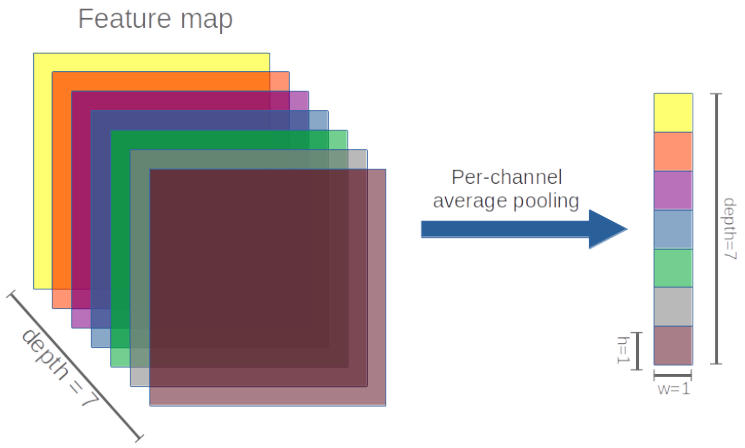


Figure 2.14: Global Average Pooling layer.

Inception module

Recall that a convolution layer is meant to learn features from a 3D object with 2 spatial dimensions (width and height) and a channel dimension. This means that each kernel in the convolution needs to learn simultaneously cross-channel and spatial correlations. The intuition behind the *Inception module* is to improve this process

by separating this two tasks, so that the cross-channel correlations and the spatial correlations can be learned separately and independently [Cho17].

A normal inception model looks at the cross-channel correlations first through a set of 3 or 4 *pointwise convolutions*[13], and then learns the spacial information in the downsampled feature image (in depth, not height and width), by means of regular convolution (usually with 3 by 3 or 5 by 5 kernels). Figure 2.15 shows a diagram of an Inveption V3 module.

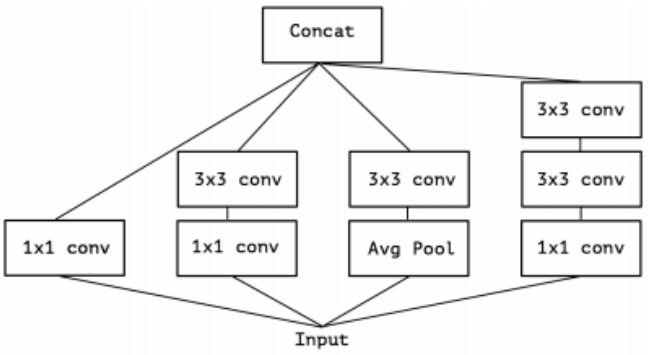


Figure 2.15: A regular Inception module (Inveption V3). Image source [Cho17].

François Chollet [Cho17], used the inception module as reference to propose the *depthwise separable convolution*, which is something between a normal convolution and a normal convolution combined/followed by a pointwise convolution. Figure 2.16 shows an *extreme* version of the inception module shown in figure 2.15. The *depthwise separable convolution* is very similar to the one shown in figure 2.16, the only difference is that the pointwise convolution is applied before the 3 by 3 convolutions instead of after.

Even though the *depthwise separable convolution* is a simplified version of the inception module, the idea and motivation behind it is the same. The *depthwise separable convolution*, and the residual block, are the main components of the *Xception* architecture [Cho17].

2.3 Interpretability Methods

In recent years, Deep Neural Networks (DNNs) have been used to solve a wide variety of problems and gained popularity. Amazing results such as those achieved by Deep Mind's Alpha Fold team, have shown the great potential DNN has to solve complex problems. However, the difficulty to interpret DNNs has become one of the main

[13] A *pointwise convolution* is a convolution with 1 by 1 kernels and stride 1.

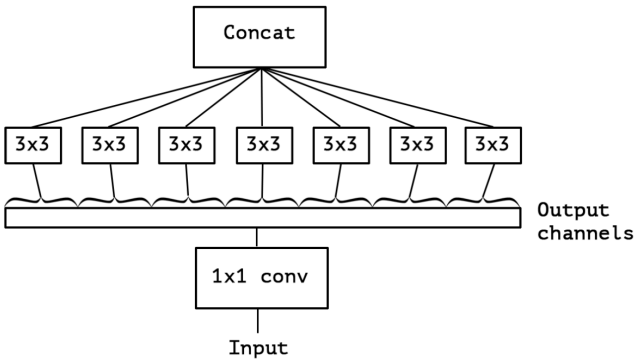


Figure 2.16: An extreme version of our Inception module. Image source [Cho17].

obstacles to their acceptance in applications where the interpretability of the model is necessary.

To understand how the DNNs predict the transcription rate (TR) of a cell, we use *Attribution Methods*. This methods are meant to measure how much each component of the input image contributes to the model's prediction by creating a *Score Map* (also known as *Importance Map, Sensitivity Map* or *Saliency Map*) of the same shape as the model's input. In particular, in this work we use a combination between Integrated Gradient (IG) [STY17] and VarGrad (VG) [Ade+20] as attribution method. In general we will denote attribution method as ϕ.

Attribution methods are not only used to interpret black-box models like DNN, the can also be used to debug models or as a sanity check to validate that the model base its prediction on the relevant features of the input.

In our case, this interpretability techniques will show us which parts of the cell image are relevant for the prediction of the TR. However, this will not just help us to interpret the results of the model, this also have the potential to help us understand unknown cellular processes.

2.3.1 Integrated Gradients

Integrated Gradient (IG) is an interpretability technique (attribution method) proposed by Sundararajan et al. [STY17], aimed to assign an importance to the input features (in our case pixels from a cell image) with respect to the model prediction. The attribution problem have been studied before in other papers [Bae+10], [SVZ13], [Shr+16], [Bin+16] and [Spr+14].

In our case, we seek to predict transcription rate (TR) given a cell image $x \in \mathbb{R}^{d \times d \times c}$, where d is the height and width of the image and c is the number of channels. Therefore, our Deep Neural Network (DNN) would be a function $f : \mathbb{R}^{d \times d \times c} \to \mathbb{R}$ and an attribution method should be a function $\phi : \mathbb{R}^{d \times d \times c} \to \mathbb{R}^{d \times d \times c}$ having an input and

output of the same shape as the model's input image.

Early interpretability methods only use gradients to assign importance to each input feature

$$\begin{aligned}\phi(f,x) &:= \nabla f(x) \\ &= \frac{\partial f}{\partial x}\end{aligned} \tag{2.12}$$

Mathematically speaking, $\phi_i(f,x)$ assign an importance score to the pixel i (out of the $d \times d \times c$ there are), representing how much it adds or subtract from the model output. However, this score maps have some drawback when they are used to interpret deep neural networks [SLL20]. Recall that the gradient with respect to the input indicate us the pixels that have the steepest local slope with respect to the model's output. This means that it only describes local changes in the input, and not the whole prediction model. Another mayor problem is saturation[14]. As the model learns the relationship between an input image and its TR, the gradient of the most important pixels will approximate to 0, i.e. the pixel's gradient saturates.

To overcome this problems, Sundararajan et al. proposed Integrated Gradient (IG) as an attribution method, where the importance of the input feature i is defined as follow

$$\phi_i^{IG}(f,x,x') := (x_i - x_i') \int_{\alpha=0}^{1} \frac{\partial f(x' + \alpha(x - x'))}{\partial x_i} d\alpha \tag{2.13}$$

Intuitively speaking, IG accumulates the input gradient when it goes from a baseline x', which should represents *absence* of information, to the actual input image x. With this, we avoid losing information about relevant pixels for the model's prediction in the importance map, even if they saturate eventually. Figure 2.17 shows an example of the image progression fed into IG. Note that the amount of information in the images is parameterized by $\alpha \in [0,1]$, and that the *absence* of information is interpreted as a black image.

For a better understanding, we can divide the IG definition as follow

$$\phi_i^{IG}(f,x,x') := \overbrace{(x_i - x_i')}^{\text{Difference from baseline}} \underbrace{\int_{\alpha=0}^{1}}_{\text{From baseline to input...}} \overbrace{\frac{\partial f(x' + \alpha(x - x'))}{\partial x_i} d\alpha}^{\text{...accumulate local gradients}} \tag{2.14}$$

The integral in equation 2.14 accumulate the gradients for the interpolated images $x' + \alpha(x - x'))$ between the baseline x' and the image x. On the other hand, the

[14] In the context of artificial neural networks, a neuron is said to be saturated when the predominant output value of a neuron is close to the asymptotic ends of the bounded activation function. This behavior can potentially damage the learning capacity of a neural network.

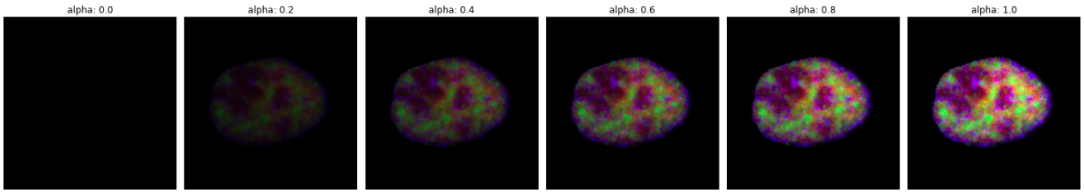


Figure 2.17: Progression from an image with no information (back image) to a normal one parameterized by α.

difference $(x_i - x_i')$ outside the integral comes from the chain rule and the fact that we are interested in integrating over the path between the baseline and the image.

IG is very simple and easy to implement, since it does not require any modification to the model and it only require some calls to the gradient operator.

The IG satisfy several properties and axioms that are addressed in detail in the paper. However, there is one axiom satisfied by IG that is of special importance for us, *completeness*. Completeness means that the value of the summed attributes will be equal to difference between the model's output when it is evaluated at the image and the model's output when it is evaluated at the baseline

$$\sum_i \phi(f, x, x')^{IG} = f(x) - f(x') \tag{2.15}$$

In practice, computing the analytic expression for the integral in equation 2.13 would be complicated, and in some cases unfeasible. However, luckily we can numerically approximate $\phi(f, x, x')^{IG}$ using a Riemann sum

$$\phi_i^{Approx\ IG}(f, x, x', m) := (x_i - x_i') \sum_{k=1}^{m} \frac{\partial f(x' + \frac{k}{m}(x - x'))}{\partial x_i} \frac{1}{m} \tag{2.16}$$

where m is number of steps for the Riemann sum approximation.

This is when the completeness axiom comes into scene, which is a good value for the parameter m? 10, 100, 500? To answer this question, we can simply apply the completeness axiom as a sanity check for the election of m. If m is good enough, then the value of $\sum_i \phi_i^{Approx\ IG}(f, x, x', m)$ should be close to $f(x) - f(x')$, or equivalently, the value of $|\sum_i \phi_i^{Approx\ IG}(f, x, x', m) - (f(x) - f(x'))|$ should be close to 0.

Figures 2.18b and 2.18c show a comparative between the gradient of a model output with respect to a cell image, and the IG. One can see that either for score maps computed using IG or vanilla gradients, the output is noisy and diffuse.

2.3.2 VarGrad

As we can see in figure 2.18c, Integrated Gradient (IG) attribution maps can be noisy and diffuse. To improve their empirical quality, Smilkov et al. [Smi+17] proposed

SmoothGrad (SG), which tends to reduce noise in practice and can be combined with other attribution map algorithms (like IG). The idea behind SG is pretty simple, given an input image x, you create a sample of similar images by adding noise, then compute the attribution map for each one of them using the algorithm you prefer (in our case IG), and take the average of the attribution maps. Although Smilkov et al. do not provide a mathematical proof of why SG reduce noise in score maps, they provide a conjecture and empirical evidence. For this work we use a slightly difference version called VarGrad (VG), proposed by Adebayo et al. [Ade+18] but inspired by SG, which takes the variance of the attribution maps instead of the mean. The reason for this choice is that Seo et al. [Seo+18] analyzed theoretically VG, and concluded that it is independent to the gradient and capture higher order partial derivatives.

In general, VG is defined as follow

$$\phi^{SG}(f, x) := Var(\phi(f, x + z_j)) \tag{2.17}$$

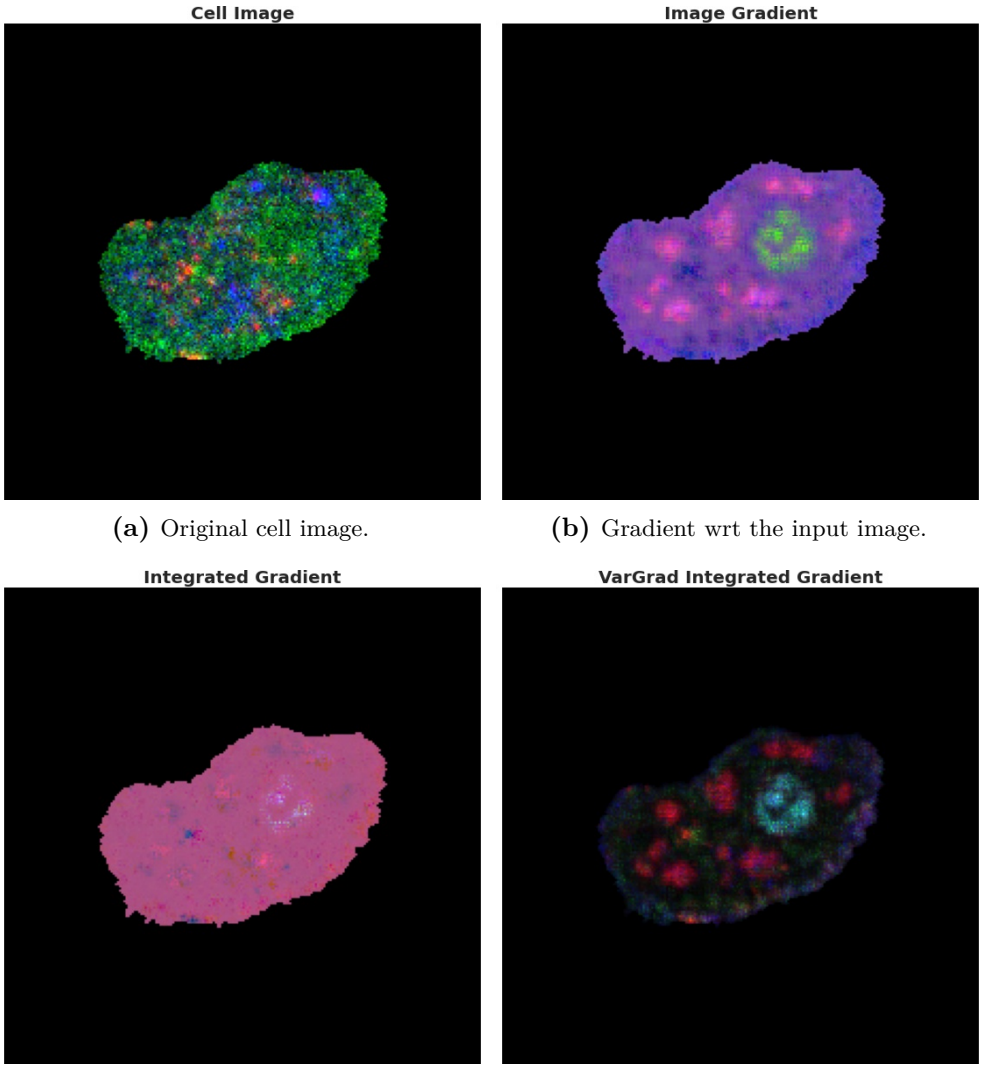
where $x \in \mathbb{R}^{d \times d \times c}$ is the input image, $f : \mathbb{R}^{d \times d \times c} \to \mathbb{R}$ a model, ϕ an attribution method to get preliminary score maps and $z_j \sim \mathcal{N}(0, \sigma^2)$, with $j \in \{1, \ldots, n\}$, are i.i.d. noise images of same shape as the input image.

Since we use IG to get preliminary score maps, in our case VG (in the subsequent defined as VarGrad Integrated Gradient (VGIG)) looks as follow

$$\phi^{SG}(f, x) := Var(\phi^{IG}(f, x + z_j, x')) \tag{2.18}$$

where $x' \in \mathbb{R}^{d \times d \times c}$ is a given baseline needed to compute the IG score maps.

Figures 2.18c and 2.18d show a comparative between IG and VGIG score maps. One can see that VGIG produces less noisy score maps than vanilla IG.



(a) Original cell image.

(b) Gradient wrt the input image.

(c) Integrated Gradient.

(d) VarGrad with Integrated Gradients.

Figure 2.18: Comparative between a cell image and the different attribution methods. All the figures show the same 3 channels taken from a cell image. a) cell image, i.e. no attribution method. b) score map using only the gradient of the model with respect to the input image. c) Integrated Gradient score map. d) VarGrad Integrated Gradient score map.

Chapter 3

The Dataset

We can interpret transcription rate (TR) as the amount of new RNA molecules inside a cell nucleus in a given period of time. By means of a fluorescent marker, it is possible to identify these new RNA molecules and thus approximate TR. But, what about the morphology of other molecules and organelles within the cell nucleus? The distribution, shape and location of molecules, proteins and organelles within the nucleus could potentially encode relevant information for cellular expression. This has been the main motivation for this work. By means of a Convolutional Neural Network (CNN), we seek to predict TR base mainly in spacial information encoded on images of cell nucleus.

In this section we introduce the process used to generate the data for this work, the multiplexed protein map (MPM) protocol. In addition to this, we introduce the preprocessing and data augmentation techniques used. These techniques aim to improve the model's training performance, prevent overfitting and remove non-relevant information from the images. With this, we seek to encourage the model to base its prediction mainly on the spatial information encoded in the images of cell nucleus.

3.1 Multiplexed Protein Maps

The amount of protein or messenger RNA (mRNA) inside a cell may not be enough to fully describe cellular function. Accordingly to Buxbaum et al. [BHS14] and Korolchuk et al. [Kor+11], cellular function can heavily depends on the specific intracellular location and interaction with other molecules and intracellular structures. Therefore, cellular expression is determined by the functional state, abundance, morphology, and turnover of its intracellular organelles and cytoskeletal structures. This means that having the ability to look at the concentration and distribution of different molecules within a cell, is an important technological achievement that can significantly leverage scientific discoveries in biomedicine. This is exactly what multiplexed protein map (MPM) allows us to do ([GHP18]). MPM are protein readouts from cell cultures, that simultaneously captures different properties of the cell, like its shape, cycle state, detailed morphology of organelles, nuclear subcompartments, etc. It also captures highly multiplexed subcellular protein maps, which can be used to identify functionally relevant single-cell states, like transcription rate (TR). These maps can also identify new

cellular states and allow quantitative comparisons of intracellular organization between single cells in different cell cycle states, microenvironments, and drug treatments [GHP18].

So, let us explain more in deept what are these MPM. Accordingly to Gabriele Gut et al. [GHP18], MPM is a nondegrading protocol that allows to capture efficiently thousands of single cell multichannel images, where each channel contains the distribution and concentration of a protein of interest inside each cell. To achieve this, the protocol is made up of different steps that will be briefly explained here.

Iterative indirect immunofluorescence imaging

The MPM protocol starts with a process called iterative indirect immunofluorescence imaging (4i) developed by the same group. The 4i is a complete protocol by itself, and it allows to capture the concentration and distribution of individual proteins in thousands of different cells in a tissue[1]. Before applying the 4i protocol, the *plate* where the cell culture is must be divided into squared sections called *wells*. Then, the 4i protocol is applied over each well and photographed in sections called *sites*.

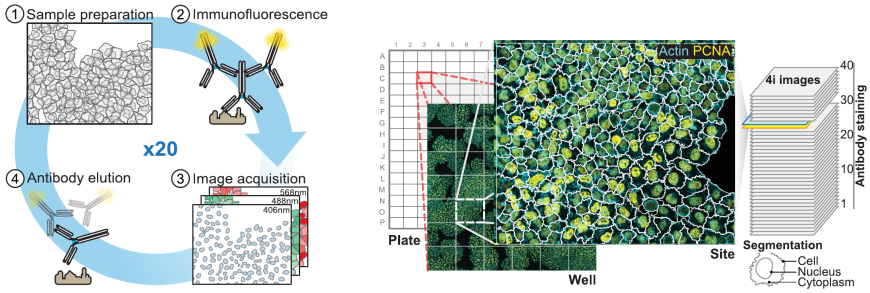
Roughly speaking, 4i works as follow

1. The selected well is prepared for the staining-elution process.
2. The well is saturated with a liquid containing *antibodies*[2] stained with a fluorescent ink (Indirect immunofluorescence (IF)), which binds to a target protein.
3. The well is exposed to a high-energy light and photographed using a light microscopy (which produces a single channel image).
4. The antibodies inside the tissue are washed-out using a chemical elution substrate.
5. Steps 2 to 4 are repeated 20 times to get 20 images of the same protein.
6. To improve the protein readouts, the 20 single channel images are projected into one by *maximum intensity projection*.

Figure 3.1a illustrates the steps of the 4i protocol that capture the saturation and distribution of a specific protein. Keep in mind that even though the 4i protocol captures sever images of the tissue, it returns an uni-channel image (step 6). Figure 3.1b shows the 4i protocol applied 40 times with different IF and over a 384-well plate, which captures the concentration and distribution of 40 different specific proteins.

[1] The tissues were made from cell cultures using the *HeLa Kyoto 184A1* cell line. HeLa is the oldest and most commonly used immortal human cell line in scientific research. The story behind it is quite interesting, so it's worth checking out.

[2] An antibody is a Y-shaped protein that can recognize and bind to a unique molecule (its antigen, e.g. another protein).



(a) Iterative indirect immunofluorescence imaging (4i) protocol.

(b) 4i protocol applied over a specific well of a plate and for 40 different IF.

Figure 3.1: Schematic representation of the 4i protocol for a single well and for 40 different fluorescent antibodies. Figure b also shows the image analysis to identify single cells and its components (nucleus and cytoplasm). Images source: [GHP18].

By the time [GHP18] was published, the 4i protocol was able to capture cell culture images with up to 40 channels without degrading the tissue, which is why MPM is called a *nondegrading* protocol.

Multiplexed single cell analysis

Once the multichannel images were generated using the 4i protocol, a series of image preprocessing and image analysis methods ([Car+06] and [Sni+12]) are applied to generate segmentation masks to identify individual cells, as well as their cytoplasm and nucleus. Figure 3.1b shows this segmentation at a cellular level, while figure 3.2 shows it also at a subcellular level. In both cases the boundaries are marked with a white contour. This single cell analysis is also used to identify cells that do not satisfy certain quality controls (like cells in the border of the image or in mitosis stage). However, this will be addressed in detail on section 3.2.

Multiplexed single-pixel analysis framework

Even though the cell cultures are now segmented into individual cells and nucleus, there is still one missing part that must be considered, and that is that cells are 3-dimensional objects. Recall that the 4i protocol saturates the cell culture with a liquid containing fluorescent antibodies. This means that the antibody can either bind to its corresponding protein inside or outside the cell nucleus. Therefore, even though that we segmented a cell into nucleus and cytoplasm, a readout assigned to the nucleus could come from a protein in the cytoplasm under or above the nucleus, and not from inside it. Fortunately, intensity readouts from proteins inside the nucleus are much higher than those in the cytoplasm. Therefore, by means of a two steps clustering

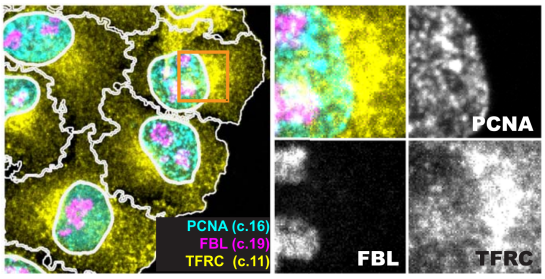


Figure 3.2: Visualization of the subcellular segmentation of a 4i protocol for 18 IF stains. The image was created by combining the readouts of 3 of this IF stains: PCNA (cyan), FBL (magenta) and TFRC (yellow). The number next to each staining label indicates their corresponding 4i acquisition cycle (4i protocol step 5). The orange rectangle and the tile at its right shows a section of the nucleus and cytoplasm of a single cell. The other 3 tiles shows the 4i readout of each of the 3 proteins. Images source: [GHP18].

approach[3], pixels can be classified accordingly to their intensity profile (figures 3.3a and 3.3b), so the source of their readout can be identified. This pixel type classification is called Multiplexed cell unit (MCU) and is illustrated in figure 3.3c. After pixels clusters (intensity profiles) where identified, the pixels whose measurement comes from the cytoplasm and not from the nucleus are removed.

FiXme Note! Finally, the nucleus of each cell is stored separately and identified with a unique id.

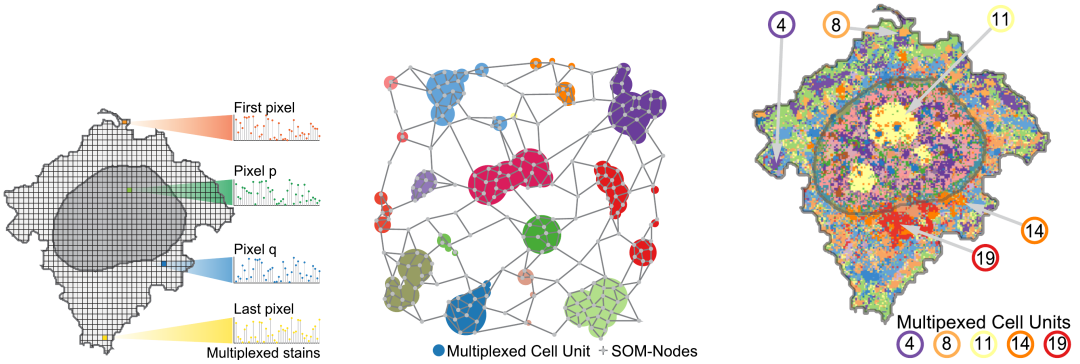
Cell cycle phase classification: G_1, S, G_2 and M phase

The MPM protocol is not only capable to capture the concentration and distribution of molecules inside thousands of cells. It can also identify the phase each cell is in, which is tightly related with the abundances and distribution of molecules inside a cell [GHP18].

Roughly speaking, cell cycle phase was determined by means of a Support Vector Machine (SVM) classifier and k-means clustering. First, a SVM classifier is trained to identify M phase cells based on the nuclear information in one of the image channels ($DAPI$[4]). Then, based on the nuclear information of channel $PCNA$, a second SVM classifier is trained to identify cells in phase S. Finally, cells in phase G_1 and G_2 are classified using a k-means algorithm, using the pixel intensity profiles of the DAPI channels excluding the cells in S and M phase. A more detailed explanation of the cell cycle classification process can be found on the dataset paper [GHP18].

[3] To identify clusters in an unsupervised manner, *Self Organizing Maps* algorithm and *Phenograph* analysis were used over a very large number of pixels sampled from a large number of single cells [GHP18].

[4] A brief description of this marker can be found on section B.1.



(a) Extraction of pixel intensities.

(b) Pixel clustering by Self Organizing Maps and Phenograph.

(c) Cell subdivision base on the multiplexed cell unit (MCU).

Figure 3.3: Figure a shows the pixel intensity extraction for a single cell. The pixel intensity is a vector containing the readout of that 2D location for each protein, one specific protein readout per entrance. Figure b shows the clusters found by Self Organizing Maps algorithm and Phenograph analysis over the pixel intensities. Figure c shows a cell masked with the clusters found by the MCU analysis. Images source: [GHP18].

Pharmacological and metabolic perturbations

To further explore the capabilities of the MPM protocol, the creators of the dataset (Gabriele Gut et al. [GHP18]) applied the MPM protocol to a cell populations that were to nine pharmacological and metabolic perturbations. The analysis reveled expected and unexpected changes in the concentration and distribution of molecules inside the cell. However, this work focused on cells without pharmacological and metabolic perturbations. This means that only cells marked as *normal* (no perturbed cells) and *DMSO*[5] (control cells) were used.

3.2 Data preprocessing

The data preprocessing consist of 4 main steps

1. The raw data processing, where raw files are converted into images.
2. The quality control, where cells that are not useful for analysis are discarded.
3. The creation of the dataset, where data is spitted into *Train, validation* and *Test* sets and stored in a way that can be used for model training efficiently.

[5]Dimethyl sulfoxide, or DMSO, is an organic compound used to dissolve test compounds in in drug discovery and design [Cus+20].

4. The image preprocessing, where the images are prepared before training the model (clipping and standardization).

In this section we explain these 4 steps. However, the implementation is discussed in the sections A.1 (for steps 1 and 2) and A.2 (for steps 3 and 4).

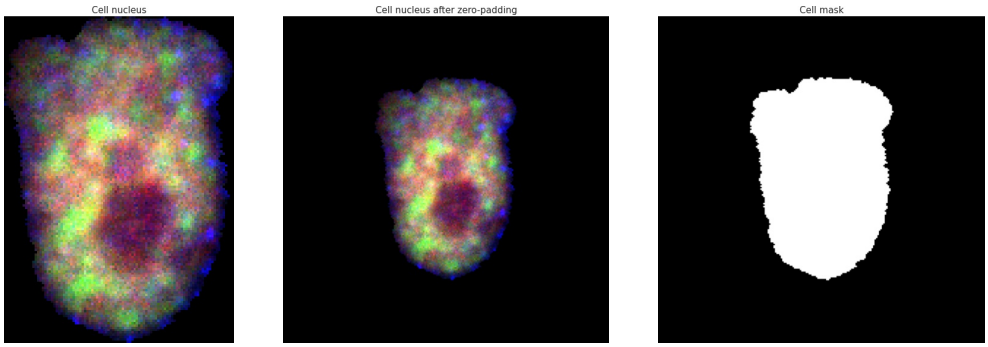
3.2.1 Raw data processing

As we mentioned in section 3.1, the multiplexed protein map (MPM) protocol is applied over section of cell cultures called *wells*. The MPM protocol will return several files for each well, containing the nuclear protein readouts of single cells, information from the subsequent analysis made to the intensities of the protein readouts, as well as information about the MPM protocol experimental setup. We do not go into details about this files and how to transform them into multichannel images of single cell nucleus. However, a brief explanation of this can be found in the appendix A.1. Appendix A.1 also show how to run the Python script that transforms the raw data into images, along with an explanation of the required parameters.

The Python script introduced on appendix A.1 extract the protein readouts from the raw data files, and use them to build multichannel images containing the nucleus of a single cell (see figure 3.4a). This means that during the reconstruction of the images, it is necessary to add black pixels (zeros) in the places where no measures were taken (like in the low corner of figure 3.4a). However, as we saw on section 2.2.2, in order to train a Convolutional Neural Network (CNN) model, all the cell images need to have a fixed size, which is denoted as I_s. For this reason, after the image is reconstructed, it is necessary to add zeros to the images borders (zero-padding) in order to make it squared and of a fixed size (see figure 3.4b). Finally, for each single cell nucleus, a *cell mask* is created to keep track of the measured and non-measured pixels (see figure 3.4c). As we can see in figure 3.4, the cell nucleus is always located in the center of the image.

The raw data processing script saves in a specified directory files containing 3 compressed NumPy arrays; 1) the multichannel image (figure 3.4b), a 3D array contains the protein readouts of the nucleus of a single cell 2) the cell mask (figure 3.4c), a 2D array that indicates the measured pixels by the MPM protocol (ones on the measured x and y coordinates and zeros otherwise) and 3) the channels average, a 1D array containing the average of the measured pixels per channel/protein. Each file is named using the unique id assigned to each single cell nucleus (`mapobject_id_cell`). The script also returns a `csv` file[6] containing the metadata of each single cell from every processed well (one row per cell and one column per cell feature). Table 3.1 shows the metadata columns that were relevant for this work.

[6]This `csv` file can be easily opened as a *Pandas DataFrame*. For more information, please refer to the official documentation.



(a) Single cell nucleus.

(b) Single cell nucleus with zero-padding.

(c) Single cell nucleus mask.

Figure 3.4: Figure a shows channels 10, 11 and 15 of the nucleus of a single cell multichannel image reconstructed form the raw data. Figure b shows image a after adding zero to the borders (zero-padding) to make it of size 224 by 224 pixels. Figure c shows the cell mask, i.e. measured pixels (in white) during the MPM protocol.

Column name	Description
`mapobject_id_cell`	ID to uniquely identify each cell among all wells
`mapobject_id`	ID to uniquely identify each cell on its well
`is_border_cell`	Binary flag, 1 if the cell is on the plate, well or site border; 0 if not
`cell_cycle`	String, `G1` if cell is in G_1 phase, `S` if cell is in synthesis phase, `G2` if cell is in G_2 phase. If `NaN`, then the cell is in mitosis phase
`is_polynuclei_184A1`	Binary flag for *184A1* cells, 1 if the cell was identified to have more than one nucleus (i.e. it is in mitosis phase); 0 if not
`is_polynuclei_HeLa`	Binary flag for *HeLa* cells, 1 if the cell was identified to have more than one nucleus (i.e. it is in mitosis phase); 0 if not
`perturbation`	String indicating the pharmacological/metabolic perturbation

Table 3.1: Relevant metadata columns.

3.2.2 Quality Control

During the transformation from raw data into images, cells that does not pass a quality control are discriminated. This quality control consist on avoiding cells that holds at least one of the following conditions

1. The cell is in mitotic phase (i.e. on metadata, either `is_polynuclei_HeLa` or `is_polynuclei_184A1` is equal to 1 or `cell_cycle` is `NaN`).

2. The cell is in the border of the plate, well or site (i.e. on metadata, `is_border_cell` is equal to 1).

The quality control is performed by the same script that transforms the raw data into multichannel images. Its implementation and execution, as well as an explanation of the required parameters, can be found on appendix A.1.

3.2.3 Dataset creation

After the raw data from all wells were processed, and mitotic and/or border cells were eliminated (quality control), we are able to build a dataset[7] that can be used efficiently to train models. We will not explain here how to create this dataset. However, a brief explanation of this can be found in the appendix A.2. Appendix A.2 also show how to run the Python script that builds this dataset, along with an explanation of the required parameters.

Even though this script can bu used to build a dataset containing all available single cell images, for this work we created a dataset containing cells without pharmacological or metabolic perturbations (i.e. cells such that in the metadata `perturbation` is ether equal to *normal* or *DMSO*). Further more, during the creation of the dataset, it is possible to filter the image channels and select the target value from the channels average vector (which is stored along with each single cell image). In this case we kept all the input channels[8], except for the channel used to calculate the target value. This means that channel 35 was excluded (`00_EU`[9]), and entrance 35 from the channel average vector (interpreted as transcription rate (TR)) was selected as target value.

Last but not least, for each cell, its mask is added at the end as an extra channel to keep track of the measured pixels. The reason why the cell mask is stored as a channel, is because it will be needed by other process latter in the pipeline (some of the data augmentation techniques, see section 3.3). However, this (and other channels) are removed before the image is used to feed the model (during and after the training process, see section 4.2).

[7] For this work we decided to use (and build) a custom TensorFlow Dataset (TFDS), which is a subclass of `tensorflow_datasets.core.DatasetBuilder` and allows to create a pipeline that can easily feed data into a machine learning model built using TensorFlow. For more information, please refer to the official documentation.

[8] The unnecessary/unwanted channels are removed during the model training/evaluation (see section 4.2). The reason why this filtering is not made during the dataset creation, is to make the dataset set more robust (i.e. to avoid the need to create a new dataset each time the input channels of the image changed).

[9] A brief description of this marker can be found on section B.1.

Table A.3 (on appendix A.2) shows the image channels in the TensorFlow Dataset (TFDS), including the name (column *Channel name*) and identifier of each immunofluorescence markers (column *Marker identifier*). Table A.3 also shows the ids corresponding to the markers in the raw data (column *Raw data id*) and in the TFDS (column *TFDS id*). *NA* means that the channel is not used/available either on the raw data or the TFDS.

Set	Size	Percentage
Train	2962	80%
Validation	371	10%
Test	370	10%
Total	3703	100%

Table 3.2: Distribution of the dataset partitions.

During the creation of the dataset, the images are also spitted into 3 sets, *Train, Validation* and *Test*, using the proportions 80%, 10% and 10% respectively. Table 3.2 shows the size of these 3 sets.

Set	Cell Cycle	Size	Percentage
Train	G_1	1652	55.77%
	S	864	29.17%
	G_2	446	15.06%
Validation	G_1	205	55.41%
	S	103	27.84%
	G_2	62	16.76%
Test	G_1	213	57.41%
	S	103	27.76%
	G_2	55	14.82%
Total	G_1	2070	55.90%
	S	1070	28.90%
	G_2	563	15.20%

Table 3.3: Distribution of the dataset partitions by cell phase (cell cycle).

Since we are dealing with cells in different phases (cell cycles), it is important that the distribution of the 3 phases is kept in the train, validation and test sets . The same must happen with the proportion of cells without pharmacological/metabolic perturbation (*Normal* cells) and control cells (*DMSO* cells). Tables 3.3 and 3.4 show respectively that these proportions are hold across the 3 sets.

FiXme Note!

Set	Perturbation	Size	Percentage
Train	Normal	2040	68.87%
	DMSO	922	31.13%
Validation	Normal	257	69.46%
	DMSO	113	30.54%
Test	Normal	260	70.08%
	DMSO	111	29.92%
Total	Normal	2557	69.05%
	DMSO	1146	30.95%

Table 3.4: Distribution of the dataset partitions by perturbation.

3.2.4 Image preprocessing

In this work we use CNNs and images of cell nucleus to predict TR. This means that there are two main features of the images that came into account when the model learns and predicts the TR, the spatial distribution of the elements in the image and the intensity of the colors. However, this work aims to explain and predict transcription based on the information encoded in the spatial distribution of proteins and organelles within the nucleus. Therefore, the image preprocessing techniques applied here should help mitigate the influence of color during training and prediction, so that the model can focus only on spatial information. For this reason, two preprocessing techniques are applied to each cell image, clipping and standardization. The clipping, as well as the standardization, are performed during the construction of the TFDS, which can be consulted in appendix A.2.

Clipping

The idea of clipping is to avoid extreme outliers to influence or leverage the model parameters during training. Figure 3.5 gives an example of this. The blue line shows a model fitted including the outliers (the two dots on the right upper corner), while the orange line a model fitted without them.

To prevent high intense pixels to influence the model, we truncate/limit the value of pixels that are above a certain threshold. This threshold is different for each image channel and is determined using the cell images belonging to the training set. For each channel, the train images are loaded and the threshold is set as the 98% percentile of the measured pixel intensities belonging to the channel. Then, using this threshold vector (one entrance per channel) all the images in the dataset (train, validation and test) are clipped. This is done before the data standardization. Finally, the clipping parameter (threshold) of each channel is stored in a metadata file, provided along with the TFDS. Figures 3.6a and 3.6b show the pixel intensity distribution of channel

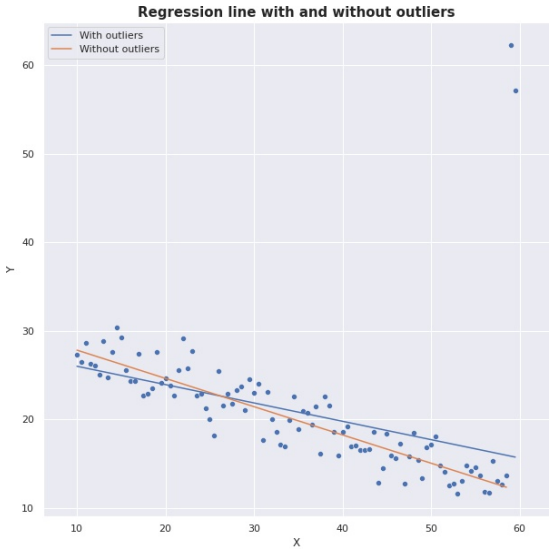


Figure 3.5: Comparison between two linear regression models, fitted with (blue line) and without (orange line) outliers.

HDAC3 before and after clipping respectively.

Standardization

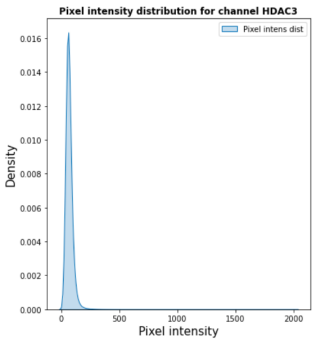
As we mentioned at the beginning of this section, to predict cell TR we seek the model to rely on spatial information, rather than the intensity of the pixels. Therefore, to reduce pixel intensity influence, we apply per-channel standardization, which is just a shift and rescaling (a linear transformation) of the original data. Standardization is also called *Z-score*, since the data is transformed using the mean μ and standard deviation σ (normal distribution parameters) of a sample, as a shift and rescaling parameters respectively. As it is done in clipping, the standardization parameters are different for each channel and are computed using the images belonging to the training set. For all the measured pixels intensities in the TFDS (i.e. for train, validation and test sets), the standardization of pixel i belonging to channel c (i.e. $z_{i,c}$), is done as follow

$$z_{i,c} = \frac{x_{i,c} - \mu_c}{\sigma_c} \tag{3.1}$$

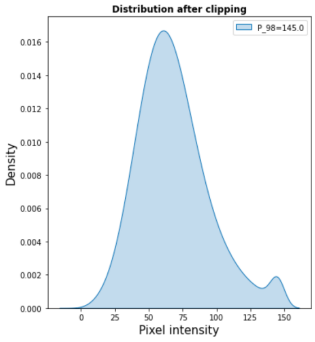
where $x_{i,c}$ is the corresponding readout i from channel c, and μ_c, σ_c are the mean and standard deviation (respectively) of channel c computed using the training images.

The standardization centers the measured pixels of each channel around 0 (see figures 3.6b and 3.6c), reducing the color correlation between channels, which also reduce pixel

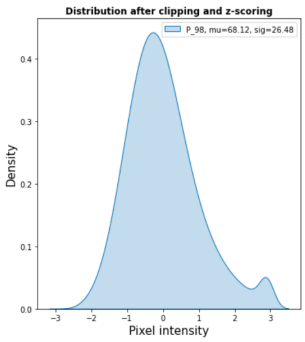
intensity influence over the model.



(a) Pixel intensity distribution.



(b) Pixel intensity distribution after clipping.



(c) Pixel intensity distribution after clipping and standardization.

Figure 3.6: Intensity distribution of measured pixels for channel HDAC3. The channel readouts were taken from the training set. Figure a) shows the distribution without any modification. Figure b) shows the distribution after applying 98% percentile clipping, while figure c) shows the distribution after applying same clipping and standardization.

Figure 3.7 shows 3 different cell nucleus sampled from the resulting TFDS. Each nucleus is in a different cell phase (G_1, S and G_2 respectively), and shows a different group of 3 markers (channels).

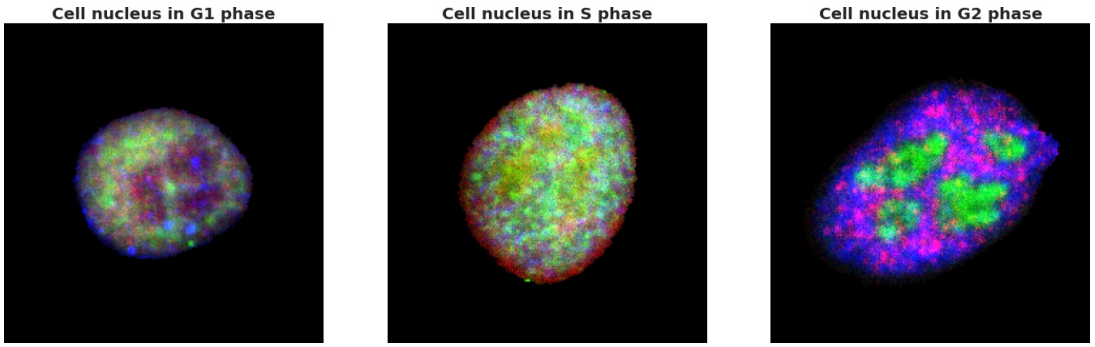


Figure 3.7: Cell nucleus in phases G_1, S and G_2 respectively. Each nucleus shows a different group of 3 markers.

3.3 Data augmentation

Data augmentation techniques are widely used to improve the results of Convolutional Neural Network (CNN) by reducing overfitting ([KSH17], [SSP+03]). This techniques improve generalization by creating new date from the existing one.

However, data augmentation techniques can not only help us to prevent overfitting, they can also be used to remove characteristics of the data that are not of interest to us. In our case, we want the model to rely on spatial information, rather than pixel intensity (color). Therefore, we implement the following data augmentation techniques, which help us to achieve this

- To remove non-relevant data features
  - Color shifting
  - Image zoom in/out
- To improve model generalization
  - Horizontal flipping
  - 90 degree rotations

This techniques are applied during training time[10] and in the same order as shown in the list above.

Nevertheless, data augmentation techniques are not only limited to the training set. As we shown above, we can divide them into two groups; 1) to remove non-relevant data features and 2) to improve model generalization. So it makes sense to apply the second group to the validation set, so we can get a better idea of how well the model is generalizing during training[11]. This means that the size of the validation set is increased by a factor of 8, by rotating each element 0, 90, 180, 270 degrees and applying (or not) a horizontal flipping. Note that the transformations performed on the validation set do not introduce any randomness to it.

3.3.1 Color shifting

The data preprocessing techniques introduced on section 3.2 (clipping and standardization), helped to reduce the influence that the intensity of the pixels, as well as the correlation between channels, have on the prediction of the model. By doing this, we encourage the model to rely more on the spatial information encoded in the images.

[10] This means that instead of generating new data and then adding it to the dataset before training, new data is generated *on the fly* during training, by applying random predefined transformations to the existing data.

[11] As always in statistics, more data equals more accurate approximations.

However, we can go a little further by shifting the pixel intensities by a random number, which would reduce even more the influence of color on the model prediction. If we sample a different random number for each channel, then the correlation between channels is also reduced. We must be careful here, since during raw data processing (see section 3.2.1) zero pixels were added to reconstruct the images. Therefore, if we add a different random number to each channel, then the non-relevant (unmeasured) pixels will have different values. Fortunately, during the creation of the TensorFlow Dataset (TFDS), for each cell we included its *cell mask* to its image as another channel (the last one). Therefore, we can use this information to randomly shift only the measured pixels. Mathematically, this means that for the channel c its $i-th$ measured pixel $x_{i,c}$, the shifted pixel $x'_{i,c}$ is defined as

$$x'_{i,c} = x_{i,c} + \eta_c \tag{3.2}$$

where $\eta_c \sim U(-a, a)$, with $c \in \{0, \ldots, C\}$, are i.i.d. random variables.

Figures 3.11a and 3.11b show a cell nucleus image before and after applying per-channel random color shifting respectively.

3.3.2 Image zoom-in/out

Size is another characteristic of the cell nucleus that could influence the output of the model. Figure 3.8 shows three cell nucleus with different sizes. However, as we already mentioned, we seek the model to predict transcription rate (TR) based on the distribution of organelles and proteins inside the nucleus (spatial information). For this reason, we randomly zoom-in/out the image to either increase or decrease (upsample or dawnsample respectively) the size of the cell nucleus inside it. This zoom is always applied over the center of the image. After that, the image either must be cropped in the center, or add zeros in the borders (padding), so the size of the zoomed image match the original size, i.e. I_s. Since this randomize the cell nucleus size, the model can not longer rely on it to make a prediction. Figures 3.11a and 3.11c show an example of this.

However, there are two things to have in mind when the size of the cell nucleus is changed, the maximum zoom-in (to avoid cutting the cell nucleus borders) and the cell nucleus size distribution.

To avoid zooming-in over a cell nucleus image too much and cut its edges, we need to determine the maximum zoom-in ratio U_{max} (which is different for every image of a cell nucleus). This can be computed as follow

$$\begin{aligned} U_{max} &:= 1 - S_{ratio} \\ &:= 1 - \frac{2d_{min}}{I_s} \end{aligned} \tag{3.3}$$

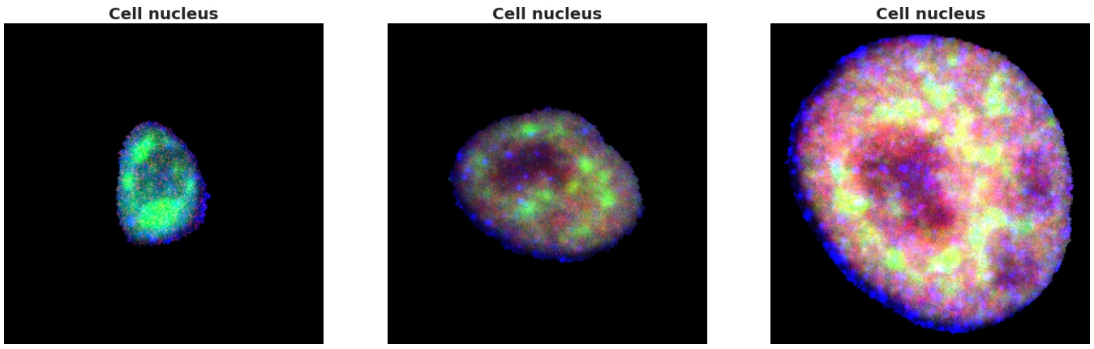


Figure 3.8: Cell nucleus with different sizes.

where $d_{min} := min\{a, b, c, d\}$ is the minimum distance between the cell nucleus and the image borders. Figure 3.9 illustrates this distances.

Intuitively, $S_{ratio} := 2d_{min}/I_s$ (cell nucleus size ratio) denotes the proportion that the cell nucleus is occupying in the image (transberzally).

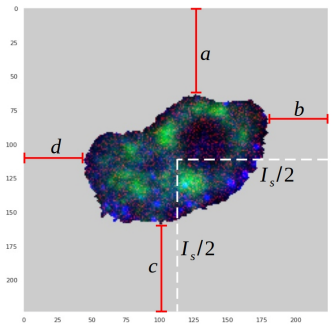


Figure 3.9: Distances needed to determine the cell size ratio. The red lines show the distance between the measured pixels of the cell nucleus (border pixels) to the 4 edges of the cell image. The white dashed lines indicates the center of the image.

The last thing that has to be considered, is the distribution of the cell nucleus sizes. Since we are randomizing them, the distribution of the new sizes must be similar to the original distribution. Fortunately, during the raw data processing (see section 3.2), the cell nucleus size ratio S_{ratio} of each cell was computed and saved in the metadata. Figure 3.10 shows the distribution of S_{ratio}. During model training, the zoom-in/out proportion is sampled considering this distribution.

3.3.3 Horizontal flips and 90 degree rotations

Since there is no sense of orientation in a cell (there is no top, bottom, left, or right), flips and rotations will not change the distribution of the data at all. For this reason,

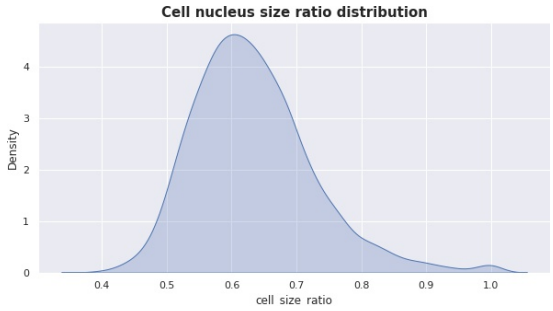


Figure 3.10: Cell nucleus size ratio S_{ratio} distribution.

we can use these transformations to simply increase the amount of data and alleviate overfitting.

For this work we used random horizontal flips and $k \times 90$ (degree) rotations, for $k \in 0, 1, 2, 3$. As we already mentioned, these transformations are applied in both the training and the validation sets.[12]. Figures 3.11a and 3.11d shows an example of a cell nucleus image after being flipped and rotated 180 degrees.

3.4 Discussion

To identify nascent RNA inside a cell nucleus, the multiplexed protein map (MPM) protocol use the 5-ethynyl uridine (EU) marker (which is then interpreted as transcription rate (TR)). However, it has been observed that this marker also binds to DNA molecules after some incubation time [JS08], [Bao+18]. As future work, another MPM dataset could be analyzed, either with a shorter or longer incubation time for the UE marker. Then, it would be interesting to validate if the results obtained with both datasets are consistent.

Besides the preprocessing techniques introduced in section 3.2 (clipping and standardization), the following approaches were also tried

- Linear scaling using the 98% percentile with and without clipping.
- Mean extraction and linear scaling using the 98% percentile with clipping (like standardization, but with the 98% percentile instead of the standard deviation).
- 49% percentile extraction and linear scaling using the 98% percentile (no clipping).

This approaches were tried at a per-channel level. However, clipping plus standardization where the prprocessing techniques that showed the best performance. Since we seek

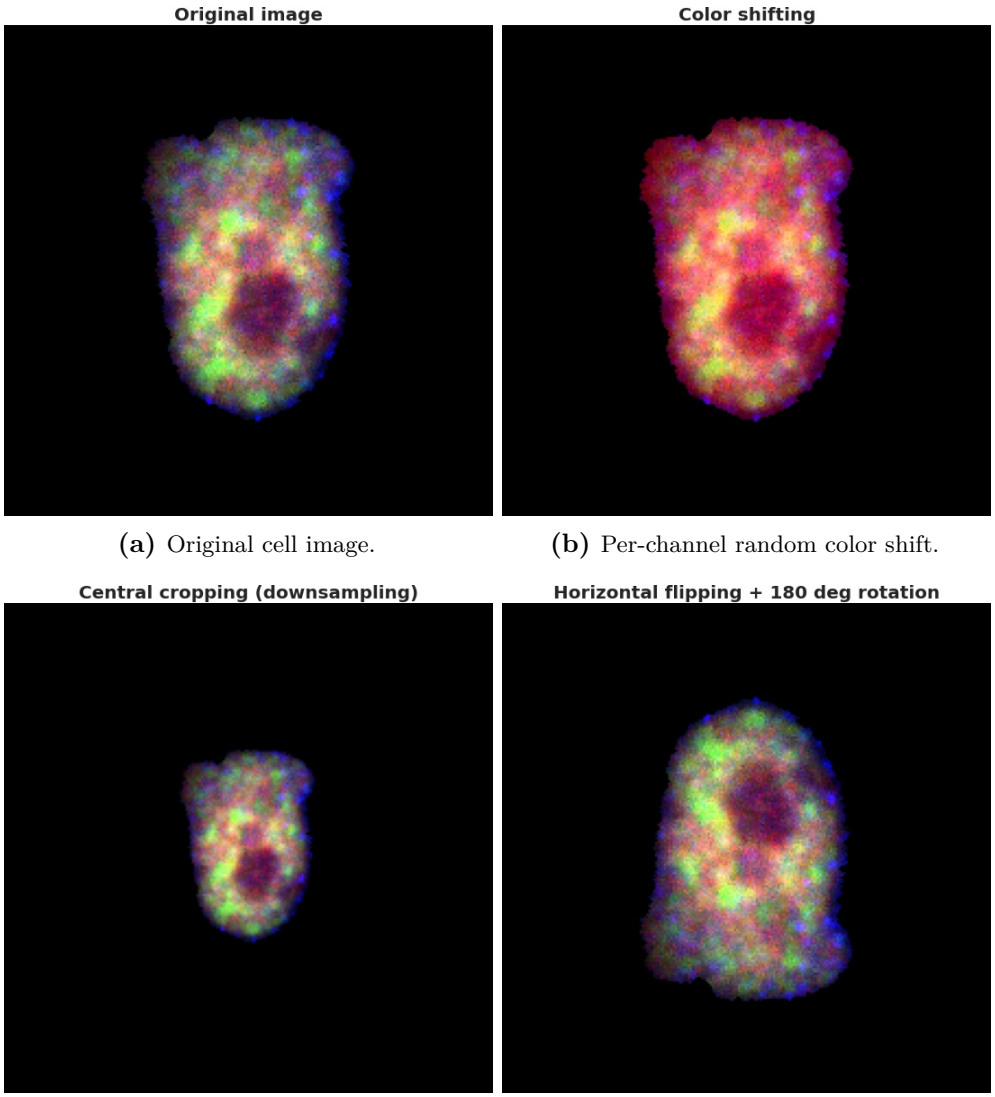
[12]The only difference is that for the validation set, the flips and rotations are applied deterministically, while for the training set they are applied randomly.

the model to predict TR base on spacial information, rather than pixel intensity/color, good performance means low Mean Absolute Error (MAE) for the Convolutional Neural Network (CNN) models, but high MAE for the linear model (since the linear model is unable to use the spatial information). This indicates that the spatial information encoded in the images of the data set has more influence on the prediction of the model than the information encoded in the colors.

Another aspect of the dataset that is worth to mention, is that more than half cells are in phase G_1 (see table 3.3), while cells in S phase are less than 30% and around 15% for G_2 cells. This causes the model to focus more on correctly predicting the TR of G_1 cells, than for cells in the other two phases. This happens because G_1 cells have more influence on the minimization of the objective function, since it is more likely that the model is fed with G_1 cells during training.

As it is shown on figure 3.12, the TR of G_1 cells is significantly lower than the TR of S and G_2 cells. This, and that cells in different phases are not in the same proportion in the dataset, could cause the model to make a biased prediction when it is fed with a S or G_2 cell. Two possible solutions to this problem are, either to add more cells in phases S and G_2 to the dataset, or to sample with replacement over the available cells, so the proportion of cells in the three different phases is the same in the dataset. Another possible solution would be to make a weight loss function based on the proportions of the cell phases, such that every phase has the same influence on it during training.

In [Smi+17], Smilkov et al. mention that the addition of random noise to the images during model training, can improve the quality of the score maps (less noisy score maps). For this reason, beside the data augmentation techniques introduced in this section, the addition of random noise was also implemented and tried. However, in practice this did not show any apparent improvement.



(a) Original cell image.

(b) Per-channel random color shift.

(c) Central cropping (downsampling).

(d) Horizontal flipping + 180 degree rotation.

Figure 3.11: Data augmentation techniques. Figure a) shows channels 10, 11 and 15 of a multichannel image without augmentation techniques. Figure b) shows image a) after applying per-channel random color shifting. Figure c) shows image a) after applying central cropping (in this case, downsampling). Figure d) shows image a) after applying horizontal flipping and 180 degree rotation (counter-clockwise).

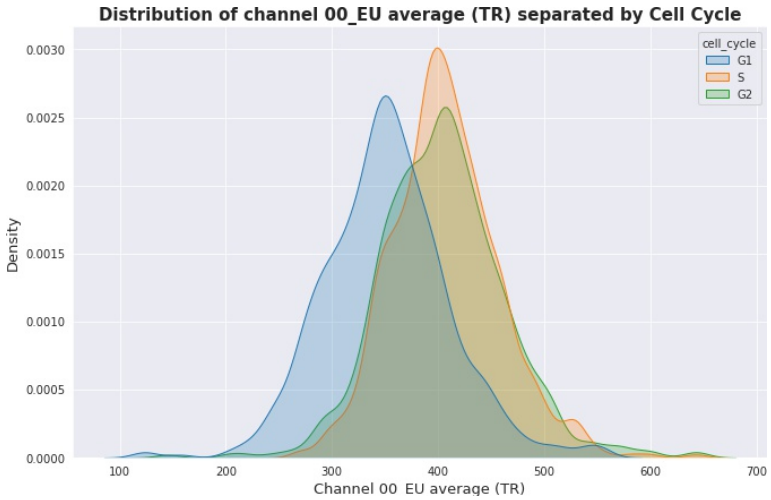


Figure 3.12: TR distribution separated by cell phase.

Chapter 4

Methodology

This chapter provides the experimental setups for each element of the workflow shown in figure 4.1. This includes the raw data processing (selected raw data and the quality control), the TensorFlow Dataset (TFDS) parameters, the data augmentation techniques (as well as their hyperparameters), the used models (architecture, loss function, optimizer and hyperparameters), the metrics used to evaluate the performance of the models and the hyperparameters corresponding to the interpretability methods.

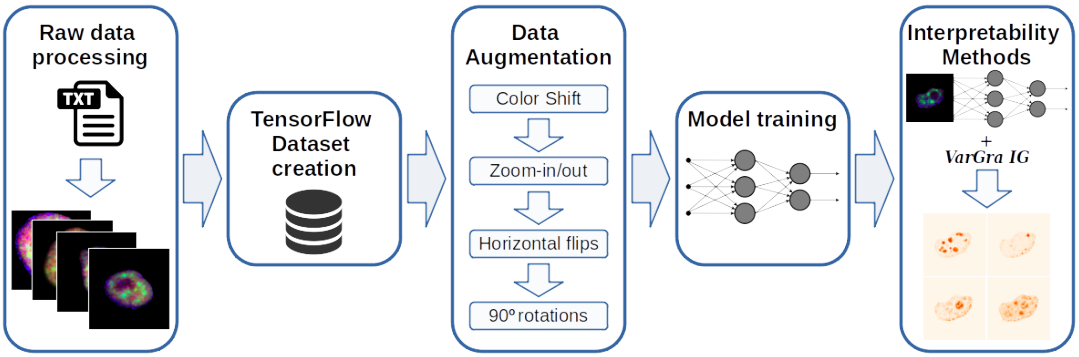


Figure 4.1: Workflow.

4.1 Dataset Setup

In this section we specify all the hyperparameters needed to execute the process explained on chapter 3. This contemplates the raw data processing, the quality control, the TensorFlow Dataset (TFDS) creation, the image preprocessing, as well as data augmentation. Here we also specify some of the experimental setups used in the multiplexed protein map (MPM) protocol.

4.1.1 Multiplexed Protein Maps

As we explained in section 3.1, the MPM protocol is capable of capturing up to 40 different proteins and molecules within a cell nucleus using fluorescent markers. The MPM data provided for this work contains 38 channels, .i.e., readouts of 38 different proteins and molecules. Table A.3 in appendix A.2, shows the marker used for each channel. Table B.1 (appendix B.1), shows an explanation of some of these markers.

The transcription rate (TR) (target variable) was not provided directly with the MPM data. For each cell, its TR was calculated by averaging the measured pixels of channel 35 (marker `00_EU`). For this reason, this channel is not used during model training/prediction. Furthermore, channel 35 contains nuclear readouts of nascent RNA in a given period of time. For the data provided, this time period was the same for each cell (30 minutes) and is specified in the *duration* columns of the metadata.

4.1.2 Data preprocessing

As we explained in section 3.2, the data preprocessing consist of 4 main steps; 1) the raw data processing, 2) the quality control, 3) the creation of the dataset and 4) the image preprocessing.

A complementary explanation of the data preprocessing parameters, as well as implementation references, can be found in the appendices A.1 and A.2.

Raw data processing

As we explained in section 3.2.3, to build the TFDS it is necessary to specify the perturbations that will be included in the dataset. For this reason, all the available wells were processed and transformed into images. This included wells exposed to pharmacological and metabolic perturbations, control wells and unperturbed wells. This allows the user to easily create new datasets without having to run the raw data processing first. Table 4.1 shows the processed wells separated by perturbation.

Perturbation type	Perturbation name	Well names
pharmacological/metabolic	CX5461	I18, J22, J09
	AZD4573	I13, J21, J14, I17, J18
	meayamycin	I12, I20
	triptolide	I10, J15
	TSA	J20, I16, J13
control	DMSO	J16, I14
unperturbed	normal	J10, I09, I11, J18, J12

Table 4.1: Well names divided by perturbation name and type.

Another hyperparameter that needs to be specified during the raw data processing, is the size of the output images I_s. This size applies to both, the width and height of the image (square images). Since some prebuilt architectures use a standard image size of 224 by 224, we define I_s as 224.

Quality Control

As it is mentioned in section 3.2.2, the quality control is meant to exclude cells with undesirable features. In our case we discriminate mitotic and border cells. The information used by the quality control is contained in the metadata of each well. Table 4.2 shows the metadata columns and the discriminated values. If a cell has any of these values, then it is excluded.

Feature	Metadata column name	Discriminated value
Cell in mitosis phace	`is_polynuclei_HeLa`	1
	`is_polynuclei_184A1`	1
	`cell_cycle`	`NaN`
Border cell	`is_border_cell`	1

Table 4.2: Discrimination characteristics for quality control.

Dataset creation and image preprocessing

As it is explained in section 3.2.3, in this work we decided to use a custom TFDS. Table 4.3 shows the parameters used to build the dataset employed in this work, together with the image preprocessing parameters.

Parameter	Description
Perturbations to be included in the dataset	*normal* and *DMSO*
Cell phases to be included in the dataset	G_1, S, G_2
Training set split fraction	0.8
Validation set split fraction	0.1
Seed	123 (for reproducibility of the train, val and test split)

Percentile	98 (for clipping / linear scaling / standardization)
Clipping flag	1
Mean extraction flag	0
Linear scaling flag	0
Standardization (z-score) flag	1
Model input channels	All of them except for channel `00_EU` (see table A.3)
Channel used to compute target variable (output channel)	`00_EU` (channel id 35, see table A.3)

Table 4.3: Parameters used to biuld TFDS and image preprocessing.

The custom TFDS created with the parameters specified in table 4.3 is called `mpp_ds_normal_dmso_z_score`.

The Python script that builds the custom TFDS, also returns a file with the image preprocessing parameters (`channels_df.cvs`) (as this is applied at a per-channel level) and information about the channels (channel name, id, etc.). It also returns another file with the metadata of each cell included in the TFDS (`metadata_df.csv`). These files are stored in the same directory as the TFDS files.

4.1.3 Data augmentation

In this section we specify the hyperparameters used for the data agumentation techniques introduced in section 3.3. Recall that the techniques are either aimed to remove non-relevant characteristics of the data (color shifting, central zoom in/out) or to improve model generalization (horizontal flipping, 90 degree rotations). Table 4.4 shows the used hyperparameters grouped by objective and technique.

Objective	Technique	Hyperparameter	Description
Remove non-relevant features	color shifting	flag	1
		distribution	uniform
		distribution mean	0
		distribution standard deviation	1
	central zoom in/out	flag	1
		mode	`random_normal`[1]

Improve generalization	horizontal flipping	flag	1
	90 degrees rotations	flag	1

Table 4.4: Parameters used for data augmentation techniques.

Even thought we specify the data augmentation hyperparameters here, in practice these are set for each model and applied during training time. However, all the models showed in this work were trained using these values. A complementary explanation can be found in appendix A.3.

In section 3.3 we mentioned that data augmentation techniques can be applied to both the training set and the validation set. However, we also mentioned that only horizontal flips and 90 degree rotations are applied for the validation set. Furthermore, for the training set these techniques are applied randomly, while for the validation set they are applied deterministically. Therefore, the hyperparameters specified in table 4.4 only apply to the training set.

4.2 Models

In this section we introduce the models, and its architecture, used in this work. We also specify all the used hyperparameters. Appendix A.3 explains briefly how to train and evaluate all the models introduced here.

In general, all the models where trained using *ReLU* as activation function for the hidden layers. Also, the identity was used as activation function for the last layer. Table 4.5, shows the other general hyperparameters.

Parameter	Description
Number of epochs	800
Early stopping patience	100
Batch size	64
TFDS name	`mpp_ds_normal_dmso_z_score`
Random seed	123
Input Channels	all channels such that its TFDS id is in $\{0, \cdots, 32\}$ (see appendix A.2, table A.3)

[1]This means that the random variable *cell nucleus size ratio* S_{ratio} (see section 3.3.2) will be sampled from a normal distribution. However, the user does not specify the parameters (mean and standard deviation). Instead, they are estimated using the information in column `cell_size_ratio` of the TFDS metadata file. Therefore, the `return_cell_size_ratio` flag must be set to 1 (True) during raw data processing, so this column is created (see section 3.2.1 and appendix A.1).

Table 4.5: Hyperparameters used in the training of all the models.

Even though that the number of epochs is specified in table 4.5, if the loss function does not improve (decrease) for more than 100 (i.e. *Early stopping patience*) epochs during training, then the training stops.

Table 4.5 also indicate the input channels to by used by the model as predictors. In section 3.2.3 we mentioned that all the image channels (with the exception of channel `00_EU`) were kept during the creation of the TensorFlow Dataset (TFDS). Moreover, since the data augmentation techniques are only applied to the measured pixels of the cell images, the cell mask was added to the image as the last channel. For this reason the channel filtering process is made inside the model. This means that after the input layer, the models have a *channel filtering layer*, which basically remove the non-selected channels, by projecting the input image from a space of shape $(bs, 224, 224, 38)$ into a lower one of shape $(bs, 224, 224, 33)$. This is done just by performing a matrix multiplication between the input batch $B \in \mathbb{R}^{bs \times 224 \times 224 \times 38}$ and a projection matrix $P \in \{0, 1\}^{38 \times 33}$ (a zero matrix with ones on the diagonal elements corresponding with to the input channels), i.e. $B_{filterd} = BP$.

All the model in this work were trained using the *Huber* loss function

$$\mathcal{L}_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{for } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases} \tag{4.1}$$

where $\delta > 0$ is the value where the Huber loss function changes from a quadratic to linear. The hyperparameter δ was set to 1 for all the models.

Huber loss function is quadratic when the error $a = |y - f(x)|$ is below the threshold δ (like the Mean Squared Error (MSE)), but linear when it is above it. This makes Huber loss less susceptible to outliers. Figure 4.2 shows a comparison between the Huber (in green) and the MSE (in blue) loss functions.

In section 2.2.1 we mentioned that we choose the Adaptive Moment Estimation (Adam) optimizer to fit the model parameters. With the exception of the *learning rate*, the used parameters were $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1e - 07$, which are the default TensorFlow hyperparameters[2]. The learning rate is specified in the section corresponding to each model.

4.2.1 Linear Model

As we already mentioned, the objective of this work is to explain cell expression using spatial information in multichannel images of cell nucleus.

[2]For more information pleases refer to the TensorFlow official documentation.

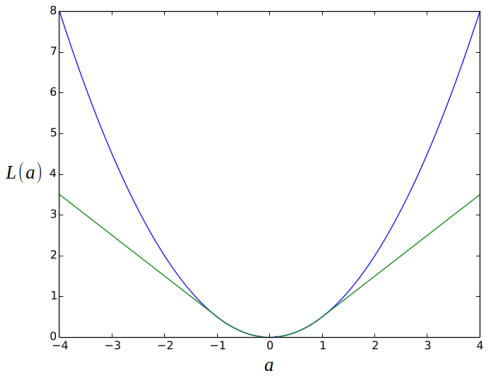


Figure 4.2: Huber (green) and the MSE (blue) loss functions. Image source [Hub].

To compare the performance of the Convolutional Neural Networks (CNNs), and have an idea about how much pixel intensity information was still contained in the data, we also fitted a *linear model*

$$y = w_0 + w_1 x_1 + \cdots + w_{33} x_{33} \tag{4.2}$$

where $x_i, \in \mathbb{R}$, for $i \in \{1, \cdots, 33\}$, is the average pixel intensity corresponding to channel i and $w_i, \in \mathbb{R}$, for $i \in \{0, \cdots, 33\}$ are the model coefficients.

The linear model architecture is specified in table 4.6. The rows of the table represent each layer of the model, which are evaluated from top to bottom. This model has only 34 free (learnable) parameters in total.

Layer	Output Shape	Number of parameters
Input	$(bs, 224, 224, 38)$	0
Channel filtering	$(bs, 224, 224, 33)$	0
Global Average Pooling	$(bs, 33)$	0
Dense	$(bs, 1)$	34

Table 4.6: Linear model architecture. The rows represent each layer of the model. The flow of the model is from top to bottom. The *bs* on the *Output Shape* column stands for *Batch size*.

The learning rate used to train the baseline model was 0.1.

4.2.2 Baseline CNN

The *Baseline* CNN architecture is specified in table 4.7. The rows of the table represent each layer of the model, which are evaluated from top to bottom. This model has 160,129 free (learnable) parameters in total.

Layer	Output Shape	Number of parameters
Input	$(bs, 224, 224, 38)$	0
Channel filtering	$(bs, 224, 224, 33)$	0
Convolution	$(bs, 224, 224, 64)$	19072
Batch Normalization	$(bs, 224, 224, 64)$	256
ReLU	$(bs, 224, 224, 64)$	0
Max Pooling	$(bs, 112, 112, 64)$	0
Convolution	$(bs, 112, 112, 128)$	73856
Batch Normalization	$(bs, 112, 112, 128)$	512
ReLU	$(bs, 112, 112, 128)$	0
Max Pooling	$(bs, 56, 56, 128)$	0
Global Average Pooling	$(bs, 128)$	0
Dense	$(bs, 256)$	33024
Batch Normalization	$(bs, 256)$	1024
ReLU	$(bs, 256)$	0
Dense	$(bs, 128)$	32896
Batch Normalization	$(bs, 128)$	512
ReLU	$(bs, 128)$	0
Dense	$(bs, 1)$	129

Table 4.7: Baseline CNN architecture. The rows represent each layer of the model. The flow of the model is from top to bottom. The *bs* on the *Output Shape* column stands for *Batch size*.

All convolution layers specified in table 4.7 used kernels of size 3 by 3 and stride of 1. Besides that, all pooling layers used a kernel of size 2 by 2 and stride of 2.

Last but not least, the learning rate used to train the baseline model was 0.0005.

4.2.3 ResNet50V2

Besides the *Baseline* CNN, we also tried the *ResNet50V2* CNN, which is a more complex (deeper) architectures than the *Baseline* model. The ResNet50V2 architecture is basically several residual blocks (see section 2.2.1), composed with convolution and pooling layers (see section 2.2.2) stacked one after another. There is a lot of literature on the ResNet50V2 architecture ([He+15], [He+16]), so we will not dive into details

here. However, the model architecture is shown in table 4.8. The raw *ResNet50V2 feature extraction*, represent the feature extraction layers (i.e., all the layers containing convolution and/or pooling layers) of the ResNet50V2[3], while the remaining rows represent the layers intended to make the final prediction. The layers are evaluated from top to bottom. This model has $24,171,777$ free (learnable) parameters in total.

Layer	Output Shape	Number of parameters
Input	$(bs, 224, 224, 38)$	0
Channel filtering	$(bs, 224, 224, 33)$	0
ResNet50V2 feature extraction	$(bs, 7, 7, 2048)$	23,612,672
Global Average Pooling	$(bs, 2048)$	0
Dense	$(bs, 256)$	524544
Batch Normalization	$(bs, 256)$	1024
ReLU	$(bs, 256)$	0
Dense	$(bs, 128)$	32896
Batch Normalization	$(bs, 128)$	512
ReLU	$(bs, 128)$	0
Dense	$(bs, 1)$	129

Table 4.8: ResNet50V2 CNN architecture. The rows represent each layer of the model. The flow of the model is from top to bottom. The *bs* on the *Output Shape* column stands for *Batch size*.

The learning rate used to train the baseline model was 0.0005.

4.2.4 Xception

As we saw in section 2.2.1, each kernel in a regular convolution layer needs to simultaneously learn spatial and cross-channel correlations. For this reason, we also tested an architecture capable of separating these two tasks, the *Xception* [Cho17].

The Xception architecture combines the idea behind the Inception module and the residual blocks (see section 2.2.1). We will not dive into details about the Xception here. However, the model architecture is shown in table 4.9. The raw *Xception feature extraction*, represent the feature extraction layers (i.e., all the layers containing convolution and/or pooling layers) of the Xception[4], while the remaining rows represent

[3]For this work, we did not implement the ResNet50V2 architecture from scratch, instead we used the pre-built model that is provided in the Keras library. For more information pleases refer to the official documentation.

[4]For this work, we did not implement the Xception architecture from scratch, instead we used the pre-built model that is provided in the Keras library. For more information pleases refer to the official documentation.

the layers intended to make the final prediction. The layers are evaluated from top to bottom. This model has 21,373,929 free (learnable) parameters in total.

Layer	Output Shape	Number of parameters
Input	$(bs, 224, 224, 38)$	0
Channel filtering	$(bs, 224, 224, 33)$	0
Xception feature extraction	$(bs, 7, 7, 2048)$	20,814,824
Global Average Pooling	$(bs, 2048)$	0
Dense	$(bs, 256)$	524544
Batch Normalization	$(bs, 256)$	1024
ReLU	$(bs, 256)$	0
Dense	$(bs, 128)$	32896
Batch Normalization	$(bs, 128)$	512
ReLU	$(bs, 128)$	0
Dense	$(bs, 1)$	129

Table 4.9: Xception CNN architecture. The rows represent each layer of the model. The flow of the model is from top to bottom. The *bs* on the *Output Shape* column stands for *Batch size*.

The learning rate used to train the baseline model was 0.001.

4.2.5 Model Metrics

To evaluate and compare the performance of the models, besides the loss function (*Huber loss*), we also used 2 other error measures

- The Mean Squared Error (MSE)

$$E_{MSE}(Y, \hat{Y}) := \frac{1}{N} \sum_{n=1}^{N} (y_i - \hat{y}_i)^2 \tag{4.3}$$

- The Mean Absolute Error (MAE)

$$E_{MAE}(Y, \hat{Y}) := \frac{1}{N} \sum_{n=1}^{N} |y_i - \hat{y}_i|^2 \tag{4.4}$$

where $Y, \hat{Y} \in \mathbb{R}^N$ are the true and predicted transcription rate (TR) values respectively.

Additionally, we also used the *Coefficient of determination* R^2, which provides the proportion of the variance in the dependent variable y that is explained by the model [SJ+60]

$$R^2 := 1 - \frac{SS_{res}}{SS_{tot}} \tag{4.5}$$

where $SS_{res} := \sum_{n=1}^{N}(y_i - \hat{y}_i)^2$ and $SS_{tot} := \sum_{n=1}^{N}(y_i - \bar{y})^2$ are the *Residual sum of squares* and the *Total sum of squares* respectively.

4.3 Interpretability Methods

There are several hyper-parameters that need to be chosen in order to compute the score map for each cell image.

For the Integrated Gradient (IG) attribution map, recall that in practice computing ϕ^{IG} could be unfeasible or computationally very expensive. However, we can approximate ϕ^{IG} by means of $\phi^{Approx\ IG}$ (see equation 2.16). Therefore, we need to define the number of steps m for the Riemann sum approximation. In section 2.3.1 we also mentioned the necessity to set a baseline image x', which should contain no information about the image, in order to compute the IG. There are several options that can be used, each one of them with different advantages and disadvantages. However, for this work we only implemented two of them: 1) a simple black image (image containing only zeros) and 2) an image filled with Gaussian noise ($\mu = 0,\ \sigma = 1$). A very good analysis on the choice of the baseline can be found in this reference [SLL20].

In section 2.3.2 we saw that for VarGrad (VG) we need to define 2 parameters, the number of noisy images n (sample size) and the standard deviation σ for the the noise distribution.

As a rule of thumbs, a sample should not be smaller than 30, so this could be a feasible option. However, since Smilkov et al. [Smi+17] showed empirically that no further improvemnt (less noise) in score maps is observed for sample sizes greater than 50, we chose this bound as sample size.

Table 4.10 shows a summary of the parameters chosen to calculate the VarGrad Integrated Gradient (VGIG) score maps.

Method	Hyperparameter	Value
IG	m	70
	x'	black image
VG	n	50
	σ	1

Table 4.10: Parameters to compute score maps.

In section 2.3.1, we mentioned that the IG algorithm holds the *Completeness Axiom*, which means that the sum of all the components of the IG attribution map must be equal to the difference between the model's output evaluated at the image and the

model's output evaluated at the baseline (see equation 2.15). This property allow us to check empirically if the number of steps m selected for the Riemann sum approximation is sufficiently large. Figure 4.3 shows that for our baseline model, a random image and $m = 70$, the completeness axiom is satisfied sufficiently well.

FiXme Note!

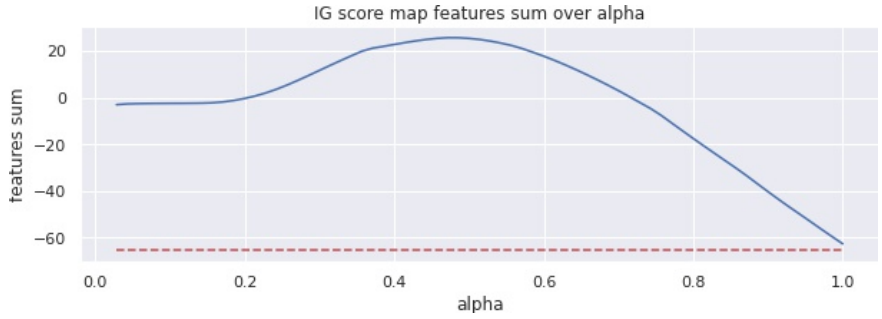


Figure 4.3: Sanity check for the number of steps m in the Riemann sum to approximate ϕ^{IG}. The red dotted line represent the difference $f(x) - f(x')$. The blue line represents the value of $\sum_i \phi_i^{Approx\ IG}(f, x, x', m)$ over α.

4.3.1 Discussion

For all the implemented architectures, L_1 and L_2 regularization was tried for both dense and convolution layers. However, in practice this did not significantly improve the generalization of the models, but it did increase the number of hyperparameters that need to be tuned (which means more models to train). For this reason the use of regularization was discarded for all the models shown on this work (L_1 and L_2 regularization strength was set to 0).

For the *ResNet50V2* and *Xception* architectures, the use of pre-trained weights and biases to initialize the model parameters (transfer learning) was also tried[5]. However, in practice this did not significantly improve the performance of the models, and therefore, was discarded.

[5]This pre-trained parameters were obtained from the Keras library (which were fitted using the ImageNet dataset [Rus+15]).

Chapter 5

Results

A small overview. Explain roughly the pipline: Preprocessing -> TDFS -> CNN -> VatGrad IG -> ROAR

Hannah's recommendations:

- Results for each method/model in subsection
- Discuss results; what does this mean? Does this prove your hypothesis? Disprove it?
- Ideally final paragraph connecting different results

here you also need to specify the early stopping epoch of each model.

5.1 Model Performance

5.1.1 Baseline values

5.1.2 Linear Model

5.1.3 Baseline CNN

5.1.4 ResNet50V2

5.1.5 Model Performance Comparative

FiXme Note!

5.2 Model Interpretation

5.3 Discussion

Model	Dataset Properties	Bias	Std	R2	MAE	MSE	Huber
Linear	color and structure	-0.63	44.68	0.49	33.08	1991.45	32.58
	structure	-2.45	54.99	0.22	41.54	3022.32	41.04
CNN	color and structure	-1.44	42.25	0.54	31.81	1782.90	31.32
	structure	-2.72	44.84	0.48	32.84	2012.85	32.35
Baseline	targets average	0	0	0	46.81	3685.75	46.31

Table 5.1: Model performance comparative. Row *Dataset Properties* indicate what information was preserved in the dataset during training. *structure* indicates that per-channel random color shifting was applied as data augmentation technique, to reduce information contained in the cell image colors. *color and structure* indicate that no random color shifting was applied. The row indicated as *baseline* contains the values for the metrics, when the model always return the average value of the targets (on the training set).

Chapter 6

Conclusion

Nothing new here, only a short recap of the project, it's results, as well as possible future work.

As future work, it would be interesting to explore other interpretability methods or go more in-depth with the current one by, for instance, exploring different baseline for Integrated Gradient and analyzing their impact on the importance maps. It also would be interesting to extend this work to the perturbed dataset and to see how the pharmacological and metabolic perturbations change score maps. To see if instead of relaying on the shape of the nucleolus and/or splicing speckles, the Convolutional Neural Network (CNN) looks at other subnuclear structures. It also would be interesting to see how the similarities between score map channels (or score maps channels and cell image channels) change.

Run the analysis with a dataset that has the same number of cells in phase G_1, S, G_2, and see if the model accuracy improve. Also validate if the results of the score maps change.

Because it has been observed that the EU marker also binds to DNA molecules after some incubation time [JS08], [Bao+18], as a future work another multiplexed protein map (MPM) dataset could be analyzed, either with a shorter or longer incubation time for the UE marker. Then, it would be interesting to validate if the results obtained with both datasets are consistent.

Hannah's recommendations:

- Summarise previous chapters and list main results
- Depending on which suits your project better, you can also do the discussion in this chapter instead of in the results chapter
- Last paragraph of the discussion should include important outlook with future impact.

Appendix A

Remarks on Implementation

This appendix contains notes about how to execute all the scripts and notebooks used in this work. It also contains information about the parameters that need to be specified for each program. All the scripts and notebooks were written in `Python` and executed over `Anaconda`. You can find information about the environment setup, packages version, etc. here.

The logic to execute any `Python` script is always the same

```
python python_script_name.py -p ./Parameters_file_name.json
```

For the `Jupyter Notebooks` you just have to open it and set the variable `PARAMETERS_FILE` with the absolute path and name of the input parameters file

```
PARAMETERS_FILE = "/path_to_file_dir/Parameters_file_name.json"
```

For each script/notebook all the needed parameters have to be specified inside its parameter file only. The format for the parameters file is always `JSON` and the parameters values are specified in a python-dictionary format. Al Binary parameters need to be specified as 1 or 0 (True or False respectively).

A.1 Raw data processing and QC implementation notes

This appendix contains implementation and technical notes about the data preprocessing process that needs to be performed before the construction of the dataset used to train the models. This process is performed by a single Pyhton script (Jupyter Notebook) and contemplate two main steps

1. The reconstruction of single cell images from the raw data (text files).

2. The discrimination of single cell images accordingly to a quality control.

As it is explained in section 3.2, the protein readout of each well are contained in several files. Here we introduce those that are relevant for this work

- `mpp.npy`: 2D NumPy array. Each row contains the protein readouts (intensities) of each pixel of the well (one column per protein). The values of this array vary from 0 to 65535 i.e. 2^{16} i.e. 2 bytes or 16 bits.
- `x.npy`/`y.npy`: 1D NumPy array. Each entrance contains the x/y coordinate of a pixel of the well protein readouts (i.e. `x.npy` and `y.npy` map the protein readouts in `mpp.npy` with a 2D plane). Accordingly with [GHP18], the size of a single channel well image is 2560x2160. Therefore, the values in `x.npy` vary between 1 and 2560 and form 1 to 2160 for `y.npy`
- `mapobject_ids.npy`: 1D NumPy array. Each entrance contains an id that maps the protein readouts in `mpp.npy` with the nucleus of a cell in the well. Each cell nucleus in the well is identified by a unique id.

Since files `mpp.npy`, `x.npy`, `y.npy` and `mapobject_ids.npy` contains different parts of the well protein readouts, the first dimension of the arrays contained in the files always has the same size.

Beside the files with protein readouts (`npy` files), each well also comes with two additional `csv` files[1] containing further information about each cell in the well

- `metadata.csv`. Contains one raw per single cell nucleus in the well. The mapping between the metadata file and the protein readouts (`npy` files), is made through the column *mapobject_id*, which uniquely identify cells (but only within the well). On the other hand, column *mapobject_id_cell* uniquely identify cells across all wells. Columns *is_polynuclei_HeLa* and *is_polynuclei_184A1* indicate if a cell is in mitosis phase. This metadata file also contains information about the experimental setup, like plate name, well name, site position, etc..
- `channels.csv`. Contains only two columns that maps the immunofluorescence marker name (column *channel_name*) and the channel id (column *id*) of the protein readouts.

The files introduced so far are specific to each well. However, we still need to introduce other 3 files which contains information about all the wells

- `secondary_only_relative_normalisation.csv`. Contains the experimental setup information related to the image capturing process. Among other information, it contains the *background value* of each channel that has to be subtracted from the protein readouts during the reconstruction of the images.
- `cell_cycle_classification.csv`. Contains the phase of each cell.

[1] These `csv` files can be easily opened as a *Pandas DataFrame*. For more information, please refer to the official documentation.

- `wells_metadata.csv`. Contains more information about the experimental setup. Among other information, it contains the pharmacological/metabolic perturbation applied to each well.

To execute the raw data processing, one has to open the Python Jupyter Notebook `MPPData_into_images_no_split.ipynb` and replace the variable `PARAMETERS_FILE` with the absolute path and name of the input parameters file before running the notebook. A sample parameter file (`MppData_to_imgs_no_split_sample.json`) is provided along with this work. It contains the parameters used for the experiments shown in this work. Table A.1 provides an explanation of some of this parameters.

JSON variable name	Description
`raw_data_dir`	Path where the directories that contain the raw data files of each well are
`perturbations_and_wells`	Dictionary. The dictionary keys must be the directories for each perturbation, while the elements (a list) must contain the directory name of each well (one list entrance per well)
`output_pp_data_path`	Path where the output folder of the notebook must be located
`output_pp_data_dir_name`	Folder name where the notebook output will be saved
`img_saving_mode`	Indicate the shape of the output images. To replicate the experiments of this work, this variable must be set to `original_img_and_fixed_size`, which means squared images of fixed size
`img_size`	Integer. High and width of the output image (squared)
`return_cell_size_ratio`	Binary. Indicate if cell size ratio (percentage of the image that is occupied by the cell nucleus measurements) must be added to the output metadata file. During the data augmentation, this information can be used to approximate the parameters of the distribution used to randomly vary the size of the cell nucleus
`background_value`	Path and name (normally `secondary_only_relative_normalisation.csv`) of the metadata file containing the per-channel background values
`subtract_background`	Binary. Indicate if background color need to be subtracted from each channel

`cell_cycle_file`	Path and name (normally `cell_cycle_classification.csv`) of the metadata file containing the phase of each cell
`add_cell_cycle_to_metadata`	Binary. Indicate if cell phase must be add to the output metadata file
`well_info_file`	Path and name (normally `wells_metadata.csv`) of the metadata file containing the information about well perturbation
`add_well_info_to_metadata`	Binary. Indicate if columns of `well_info_file` must be add to the output metadata file
`filter_criteria`	List containing the metadata columns names that will be used in the quality control. For this work ["is_border_cell", "is_polynuclei_184A1", "is_polynuclei_HeLa", "cell_cycle"] was used
`filter_values`	List containing the filtered values for the columns indicated in `filter_criteria`. For this work [1, 1, 1, "NaN"] was used
`aggregate_output`	Indicate how to project each image channel into a number. Must be equal to "avg" (average)
`project_into_scalar`	Binary. Indicate if the channel scalar projection must be add to the output metadata file

Table A.1: Parameters to perform the raw data processing.

Roughly speaking, the notebook iterates over the specified wells sequentially. This means that for each well the notebook

1. Reads the well metadata file `metadata.csv` and merge it with the general metadata files, `cell_cycle_classification.csv` and `wells_metadata.csv`.
2. Performs the quality control and select the ids (*mapobject_id_cell*) that were approved.
3. Converts[2] and saves the selected ids using the well protein readouts files `mpp.npy`, `x.npy`, `y.npy`, `mapobject_ids.npy` and the general file `secondary_only_relative_normalisation.csv`.

The notebook also saves at the end a general metadata file (`csv` file), containing the metadata of all the processed wells.

[2]The library `mpp_data_V2.py` used to perform the raw data transformation, is almost entirely based on Dr. Hannah Spitzer library `mpp_data.py`. I thank the Dr. Spitzer for providing me with her library for this work.

A.2 TensorFlow Dataset and image preprocessing implementation notes

After the raw data was processed and converted into images of single cell nucleus (see section 3 and appendix A.1), it is possible to build a TensorFlow Dataset (TFDS) data can easily end efficiently feed data into a model built in TensorFlow. A TensorFlow Dataset is build by writing a subclass of the `tensorflow_datasets.core.DatasetBuilder` class (for more information, please refer to the official documentation), and there are some steps that need to be followed to do so. The easiest way to build a TFDS, is by running the bash script `Create_tf_dataset.sh`, which executes this steps. The script needs to be executed (and located) in the same directory where the folder containing the Python code to build the dataset is

```
./Create_tf_dataset.sh -o /Path_to_save_TFDS -n ↷
    ↪ Folder_name_containing_the_TFDS_builder_code -p ↷
    ↪ /Path_to_parameters_files/parameters_file.json -e ↷
    ↪ my_conda_env_name
```

where the flag `-o` indicates the path where TFDS will be located after it is built, `-n` the name of the directory (not the path, the folder name in the same directory as the script) containing the python (builder) code for required dataset, `-p` the absolute path and name of the input parameters file[3] and `-e` the name of the Anaconda environment used to build the TFDS. The specified Anaconda environment is necessary not just to build the TFDS, but also to register it in the environment. If a TFDS is not registers in an Anaconda environment, the `tensorflow_datasets`[4] library will not find it, and the user will not be able to call it and use it. Therefore, to register a custom TFDS in another environment, one just have to execute the `Create_tf_dataset.sh` script specifying the new environment using the `-e` flag. If the TFDS was already built by another environment, python will just register the dataset under the new environment and it will not build it again.

Table A.2 provides an explanation of the variables contained in the parameters file.

JSON variable name	Description
`data_source_parameters`	Path where the parameters file used in the raw data processing is (see appendix A.1). Several parameters from this file are used to build the TFDS
`perturbations`	A list containing the names of the perturbations to be included in the TFDS. For instance, ["normal", "DMSO"]

[3] This file needs to be `JSON` format and located in a directory named `Parameters`, which needs to be located inside the directory specified by the `-n` flag.

[4] See the documentation here.

`cell_cycles`	A list containing the names of the cell phases to be included in the TFDS. For instance, ["G1", "S", "G2"]
`train_frac`	Scalar between 0 and 1. Proportion of the data to include in the train set
`val_frac`	Scalar between 0 and 1. Proportion of the data to include in the validation set. Proportion of the data to include in the test set is 1 - `train_frac` - `val_frac`
`seed`	Scalar. For reproducibility of the train, val and test split
`percentile`	Scalar between 0 and 100. Percentile used in clipping and/or linear scaling and/or standardization
`apply_clipping`	Binary. If 1, per-channel clipping is applied using the channel percentile
`apply_mean_extraction`	Binary. If 1, per-channel mean shift is applied using the channel mean
`apply_linear_scaling`	Binary. If 1, per-channel scaling is applied using the channel percentile
`apply_z_score`	Binary. If 1, per-channel standardization is applied using the channel parameters
`input_channels`	List containing the name of the channels (elements of the column *Marker identifier* of table A.3) to be included in the images contained in the TFDS
`output_channels`	List of only ONE element containing the name of the channel to be used as the target variable (its protection, i.e. the channel average)

Table A.2: Parameters to perform the raw data processing.

As it is shown in table A.2, the parameter `input_channels` specifies the channels that will be included in the TFDS images (see table A.3). However, to avoid building a new dataset every time we change the input channels, all the channels are included here and then filtered in the model (see section 3.2.3 for a more detailed explanation).

A sample parameter file (`tf_dataset_parameters_sample.json`) is provided along with this work. It contains the parameters used in the Python script `MPP_DS_normal_DMSO_z_score.py`, to build the dataset `mpp_ds_normal_dmso_z_score` used to train the models in this work.

Channel name	Marker identifier	Raw data id	TFDS id
DAPI	00_DAPI	0	0
H2B	07_H2B	1	1
CDK9_pT186	01_CDK9_pT186	2	2
CDK9	03_CDK9	3	3
GTF2B	05_GTF2B	4	4
SETD1A	07_SETD1A	5	5
H3K4me3	08_H3K4me3	6	6
SRRM2	09_SRRM2	7	7
H3K27ac	10_H3K27ac	8	8
KPNA2_MAX	11_KPNA2_MAX	9	9
RB1_pS807_S811	12_RB1_pS807_S811	10	10
PABPN1	13_PABPN1	11	11
PCNA	14_PCNA	12	12
SON	15_SON	13	13
H3	16_H3	14	14
HDAC3	17_HDAC3	15	15
KPNA1_MAX	19_KPNA1_MAX	16	16
SP100	20_SP100	17	17
NCL	21_NCL	18	18
PABPC1	01_PABPC1	19	19
CDK7	02_CDK7	20	20
RPS6	03_RPS6	21	21
Sm	05_Sm	22	22
POLR2A	07_POLR2A	23	23
CCNT1	09_CCNT1	24	24
POL2RA_pS2	10_POL2RA_pS2	25	25
PML	11_PML	26	26
YAP1	12_YAP1	27	27
POL2RA_pS5	13_POL2RA_pS5	28	28
U2SNRNPB	15_U2SNRNPB	29	29
NONO	18_NONO	30	30
ALYREF	20_ALYREF	31	31
COIL	21_COIL	32	32
BG488	00_BG488	33	33
BG568	00_BG568	34	34
EU	00_EU	35	NA
SRRM2_ILASTIK	09_SRRM2_ILASTIK	36	35
SON_ILASTIK	15_SON_ILASTIK	37	36
Cell mask	NA	NA	37

Table A.3: Image channels. Column *Raw data id* shows the channel id used in the raw data, while column *TFDS id* shows the channel id used in the TensorFlow dataset.

A.3 Model training implementation notes

This appendix is intended to provide a brief explanation of how to run the Python script (Jupyter Notebook) responsible for training the models used in this work. In addition, here we also provide a short explanation of the parameter file that must be specified to train any model.

Since data augmentation techniques can be selected independently for each trained model, their corresponding hyperparameters are also explained here.

The Jupyter Notebook responsible for training the models is the one that requires the largest number of parameters. However, the function `set_model_default_parameters` (in the `Utils.py` library) provides default values for all the parameters. Therefore, if some hyperparameter is not specified here or in section 4.2, then the value used was the one specified in that function.

To train a model, one has to open the Python Jupyter Notebook `Model_training_class.ipynb` and replace the variable `PARAMETERS_FILE` with the absolute path and name of the input parameters file before running the notebook. A sample parameter file (`Train_model_sample.json`) is provided along with this work. It contains the parameters used to train the *Baseline* Convolutional Neural Network (CNN) (see section 4.2.2), using the data augmentation techniques specified in section 4.1.3. Table A.4 provides an explanation of some of the model training parameters, while table A.5 an explanation of some of the data augmentation parameters. Although the training and data augmentation parameters are specified in separate tables, they must be in the same `JSON` parameter file (and also as items in the same dictionary).

JSON variable name	Description
`model_name`	Name of the architecture to be trained. Available: `baseline_CNN`, `ResNet50V2`, `Xception`, `Linear_Regression`
`pre_training`	Binary, whether or not use pretrained weights and biases as initial parameters. Only available for `ResNet50V2` or `Xception` architectures
`dense_reg`	$[L_1, L_2]$, where L_1 and L_2 are the regularization strengths for the dense layers weights
`conv_reg`	$[L_1, L_2]$, where L_1 and L_2 are the regularization strengths for the convolution layers weights
`bias_l2_reg`	L_2 regularization strengths for convolution and dense layers biases
`number_of_epochs`	Maximum number of epochs to train
`early_stop_patience`	For early stopping. Specify how many epochs at most the model can train without decreasing the loss function before stopping the training

`loss`	Loss function name. Available: `mse`, `huber`, `mean_absolute_error`
`learning_rate`	Learning rate for Adam optimizer
`BATCH_SIZE`	Batch size
`model_path`	Path to save the models and checkpoints
`clean_model_dir`	Binary, whether or not to delete the content of the directory specified by `model_path`
`tf_ds_name`	Name of the TFDS to be used during training
`local_tf_datasets`	Local path where the TFDSs are stored
`input_channels`	List containing the name of the channels (elements of the column *Marker identifier* of table A.3) to be included in the images contained in the TensorFlow Dataset (TFDS)
`shuffle_files`	Binary, whether or not to shuffle the dataset at the beginning of each epoch
`seed`	Random seed to reproduce the shuffling of the TFDS

Table A.4: Model training parameters.

JSON variable name	Description
`random_horizontal_flipping`	Binary, whether or not to perform random horizontal flips on the training set
`random_90deg_rotations`	Binary, whether or not to perform random 90deg rotations on the training set
`CenterZoom`	Binary, whether or not to perform random center zoom-in/out on the training set
`CenterZoom_mode`	Zoom proportion R.V. distribution. Available: `random_normal`, `random_uniform`
`Random_channel_intencity`	Binary, whether or not to perform per-channel random color shifting on the training set
`RCI_dist`	Distribution of random color shifts. Available: `uniform`, `normal`. If `uniform` distribution selected ($U(-a, a)$), then $a = \mu + 3\sigma$
`RCI_mean`	Mean μ for the distribution specified by `RCI_dist`
`RCI_stddev`	Standard deviation σ for the distribution specified by `RCI_dist`
`Random_noise`	Binary, whether or not to add random normal noise ($N(0, \sigma)$) on the training set
`Random_noise_stddev`	Standard deviation corresponding to the normal distribution of random noise

Table A.5: Data augmentation parameters.

A.4 VarGrad IG implementation notes

In order to generate the VarGrad Integrated Gradient score maps, you must execute the python script `get_VarGradIG_from_TFDS.py` specifying the parameters file

```
python get_VarGradIG_from_TFDS.py -p ./Parameters_file_name.json
```

Table A.6 show all the parameters that need to be specified to execute `get_VarGradIG_from_TFDS.py` successfully.

Hyperparam	JSON variable name	Notes
m	`IG_m_steps`	Number of steps to approximate Integrated Gradient (IG)
x'	`IG_baseline`	Baseline image for IG. Available: "black" for a simple black image and "noise" for an image filled with Gaussian noise ($\mu = 0,\ \sigma = 1$)
n	`VarGrad_n_samples`	Number of noisy images to compute VarGrad (VG)

Table A.6: Parameters to compute score maps.

Appendix B

General Remarks

This appendix contains general remarks relevant for this work, like a small explanation of the fluorescent markers used for the multiplexed protein map (MPM) protocol.

B.1 Indirect Immunofluorescence markers description

In order to capture the distribution and amount of proteins inside a cell nucleus, the multiplexed protein map (MPM) protocol use a set of fluorescent markers called Indirect immunofluorescence (IF). Table B.1 shows the name of some of this marker, its identifier used in the parameters files for the implementation, and a small description of it.

Marker name	Marker identifier	Description
DAPI	`00_DAPI`	*4',6-Diamidino-2-Phenylindole*, or DAPI, is a fluorescent stain that binds strongly to adenine–thymine-rich regions in DNA [Kap95]
PCNA	`14_PCNA`	*Proliferating Cell Nuclear Antigen*, or PCNA, is a DNA clamp that acts as a processivity factor for *DNA polymerase* δ[1]in eukaryotic cells and is essential for replication [KLW05]
EU	`00_EU`	*5-Ethynyl Uridine*, or EU, is a molecule that binds to newly transcribed RNA [JS08]. This means that EU can be used to detect RNA synthesis in cells and/or predict transcription rate (TR)

Table B.1: Indirect immunofluorescence markers description. The first column shows the markers name, the second the identifier used on the implementation (parameters file) and the third a brief description of it.

[1]DNA polymerase delta, or DNA Pol δ, is an enzyme complex found in eukaryotes that is involved in DNA replication and repair.

List of Figures

List of Tables

Acronyms

4i iterative indirect immunofluorescence imaging. 30–32, 78

Adam Adaptive Moment Estimation. 13, 54

ANN Artificial Neural Network. 10–15, 17, 18, 21, 77

CNN Convolutional Neural Network. 2, 10, 18, 20, 29, 34, 38, 41, 45, 55–58, 64, 72, 81

DNN Deep Neural Network. 22, 23

GD Gradient Descent. 13

IF Indirect immunofluorescence. 30–32, 75, 78, 82

IG Integrated Gradient. 3, 23–27, 59, 64, 74, 78

MAE Mean Absolute Error. 45, 58

MCU multiplexed cell unit. 32, 33, 78

ML Machine Learning. 10

MLP Multilayer Perceptron. 12, 14

MPM multiplexed protein map. 29–35, 44, 49, 50, 64, 75, 78

mRNA messenger RNA. 6, 8, 9, 29, 77

MSE Mean Squared Error. 54, 55, 58, 79

pre-mRNA pre-messenger RNA. 6–9, 77

ReLU Rectified Linear Unit. 11, 77

SG SmoothGrad. 26

SGD Stochastic Gradient Descent. 13

SVM Support Vector Machine. 32

TFDS TensorFlow Dataset. 36–40, 42, 49–54, 69, 70, 73, 81

TR transcription rate. 8–12, 23, 24, 29, 36, 38, 39, 42, 44, 45, 47, 50, 58, 75, 79

VG VarGrad. 3, 23, 26, 59, 74

VGIG VarGrad Integrated Gradient. 26, 27, 59, 78

Bibliography

[Ade+18] J. Adebayo, J. Gilmer, I. Goodfellow, and B. Kim. *Local Explanation Methods for Deep Neural Networks Lack Sensitivity to Parameter Values*. 2018. arXiv: 1810.03307 [cs.CV].

[Ade+20] J. Adebayo, J. Gilmer, M. Muelly, I. Goodfellow, M. Hardt, and B. Kim. *Sanity Checks for Saliency Maps*. 2020. arXiv: 1810.03292 [cs.CV].

[Bae+10] D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen, and K.-R. Müller. "How to Explain Individual Classification Decisions". In: *Journal of Machine Learning Research* 11.61 (2010), pp. 1803–1831.

[Bao+18] X. Bao, X. Guo, M. Yin, M. Tariq, Y. Lai, S. Kanwal, J. Zhou, N. Li, Y. Lv, C. Pulido-Quetglas, et al. "Capturing the interactome of newly transcribed RNA". In: *Nature methods* 15.3 (2018), pp. 213–220.

[BS00] R. G. Bartle and D. R. Sherbert. *Introduction to real analysis*. Vol. 2. Wiley New York, 2000.

[Ber+15] J. M. Berg, J. L. Tymoczko, G. J. G. Jr., and L. Stryer. *Biochemistry*. English. W. H. Freeman, 2015. ISBN: 1464126100.

[Bin+16] A. Binder, G. Montavon, S. Bach, K. Müller, and W. Samek. "Layer-wise Relevance Propagation for Neural Networks with Local Renormalization Layers". In: *CoRR* abs/1604.00825 (2016). arXiv: 1604.00825.

[Bis06] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[BJ] T. Brown and T. B. (Jnr). *Simple representation of the gene expression process*. https://www.atdbio.com/content/14/Transcription-Translation-and-Replication. Online; accessed 2021-03-29.

[BHS14] A. R. Buxbaum, G. Haimovich, and R. H. Singer. "In the right place at the right time: visualizing and understanding mRNA localization". In: *Nature Reviews Molecular Cell Biology* 16.2 (2014), pp. 95–109.

[Car+06] A. E. Carpenter, T. R. Jones, M. R. Lamprecht, C. Clarke, I. H. Kang, O. Friman, D. A. Guertin, J. H. Chang, R. A. Lindquist, J. Moffat, P. Golland, and D. M. Sabatini. "CellProfiler: image analysis software for identifying and quantifying cell phenotypes". In: *Genome Biology* 7.10 (2006), R100. ISSN: 1474-760X.

[Cho17] F. Chollet. *Xception: Deep Learning with Depthwise Separable Convolutions*. 2017. arXiv: 1610.02357 [cs.CV].

[Cus+20] T. T. Cushnie, B. Cushnie, J. Echeverría, W. Fowsantear, S. Thammawat, J. L. Dodgson, S. Law, and S. M. Clow. "Bioprospecting for antibacterial drugs: A multidisciplinary perspective on natural product source material, bioassay selection and avoidable pitfalls". In: *Pharmaceutical Research* 37.7 (2020), pp. 1–24.

[Cyb89] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[Fun89] K.-I. Funahashi. "On the approximate realization of continuous mappings by neural networks". In: *Neural networks* 2.3 (1989), pp. 183–192.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[GHP18] G. Gut, M. D. Herrmann, and L. Pelkmans. "Multiplexed protein maps link subcellular organization to cellular states". In: *Science* 361.6401 (2018). ISSN: 0036-8075. eprint: https://science.sciencemag.org/content/361/6401/eaar7042.full.pdf.

[He+15] K. He, X. Zhang, S. Ren, and J. Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[He+16] K. He, X. Zhang, S. Ren, and J. Sun. *Identity Mappings in Deep Residual Networks*. 2016. arXiv: 1603.05027 [cs.CV].

[HSW89] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[Hub] *Huber and MSE loss plot*. https://en.wikipedia.org/wiki/Huber_loss. Online; accessed 2021-04-26.

[IS15] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[JS08] C. Y. Jao and A. Salic. "Exploring RNA transcription and turnover in vivo by using click chemistry". In: *Proceedings of the National Academy of Sciences* 105.41 (2008), pp. 15779–15784.

[JD+13] W. JD, B. TA, B. SP, G. AA, L. M, and L. RM. *Molecular Biology of the Gene*. English. Pearson, 2013. ISBN: 9780321762436.

[Kap95] J. Kapuscinski. "DAPI: a DNA-specific fluorescent probe". In: *Biotechnic & Histochemistry* 70.5 (1995), pp. 220–233.

[Ker99] J. Kerr. *Atlas of functional histology*. English. Mosby International, 1999. ISBN: 0723430721.

[KB14] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[KLW05] J. Kisielewska, P. Lu, and M. Whitaker. "GFP—PCNA as an S-phase marker in embryos during the first and subsequent cell cycles". In: *Biology of the Cell* 97.3 (2005), pp. 221–229.

[Kor+11] V. I. Korolchuk, S. Saiki, M. Lichtenberg, F. H. Siddiqi, E. A. Roberts, S. Imarisio, L. Jahreiss, S. Sarkar, M. Futter, F. M. Menzies, C. J. O'Kane, V. Deretic, and D. C. Rubinsztein. "Lysosomal positioning coordinates cellular nutrient responses". In: *Nature Cell Biology* 13 (2011), pp. 453–460.

[KSH17] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[PO+13] J. E. Pérez-Ortín, D. A. Medina, S. Chávez, and J. Moreno. "What do you mean by transcription rate?" In: *BioEssays* 35.12 (2013), pp. 1056–1062. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/bies.201300057.

[Rui] M. Ruiz. *Animal eukaryotic cell diagram*. https://en.wikipedia.org/wiki/Cell_(biology). Online; accessed 2021-03-29.

[Rus+15] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252.

[Seo+18] J. Seo, J. Choe, J. Koo, S. Jeon, B. Kim, and T. Jeon. "Noise-adding Methods of Saliency Map as Series of Higher Order Partial Derivative". In: *CoRR* abs/1806.03000 (2018). arXiv: 1806.03000.

[Shr+16] A. Shrikumar, P. Greenside, A. Shcherbina, and A. Kundaje. "Not Just a Black Box: Learning Important Features Through Propagating Activation Differences". In: *CoRR* abs/1605.01713 (2016). arXiv: 1605.01713.

[SSP+03] P. Y. Simard, D. Steinkraus, J. C. Platt, et al. "Best practices for convolutional neural networks applied to visual document analysis." In: *Icdar*. Vol. 3. 2003. Citeseer. 2003.

[SVZ13] K. Simonyan, A. Vedaldi, and A. Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps." In: *CoRR* abs/1312.6034 (2013).

[Smi+17] D. Smilkov, N. Thorat, B. Kim, F. B. Viégas, and M. Wattenberg. "SmoothGrad: removing noise by adding noise". In: *CoRR* abs/1706.03825 (2017). arXiv: 1706.03825.

[Sni+12] B. Snijder, R. Sacher, P. Rämö, P. Liberali, K. Mench, N. Wolfrum, L. Burleigh, C. C. Scott, M. H. Verheije, J. Mercer, et al. "Single-cell analysis of population context advances RNAi screening at multiple levels". In: *Molecular systems biology* 8.1 (2012), p. 579.

[Spr+14] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. "Striving for Simplicity: The All Convolutional Net". In: (Dec. 2014).

[SJ+60] R. G. Steel, H James, et al. *Principles and procedures of statistics: with special reference to the biological sciences*. Tech. rep. 1960.

[SLL20] P. Sturmfels, S. Lundberg, and S.-I. Lee. "Visualizing the Impact of Feature Attribution Baselines". In: *Distill* (2020). https://distill.pub/2020/attribution-baselines.

[STY17] M. Sundararajan, A. Taly, and Q. Yan. *Axiomatic Attribution for Deep Networks*. 2017. arXiv: 1703.01365 [cs.LG].

[Vil] M. R. Villarreal. *Transcription steps*. https://www.ck12.org/biology/transcription/lesson/Transcription-of-DNA-to-RNA-BIO/. Online; accessed 2021-03-29.

[Wik21] Wikipedia. *Primary transcript — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Primary%20transcript&oldid=1005412823. [Online; accessed 29-March-2021]. 2021.

[Zha+88] W. Zhang et al. "Shift-invariant pattern recognition neural network and its optical architecture". In: *Proceedings of annual conference of the Japan Society of Applied Physics*. 1988.

List of Corrections