Cell Identification from Microscopy Images using Deep Learning on Automatically Labeled Data

FREDRIK SALOMON-SÖRENSEN
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Cell Identification from Microscopy Images using Deep Learning on Automatically Labeled Data

Fredrik Salomon-Sörensen
fr1824sa-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor:
Roland Nilsson, Karolinska Institutet
Amir Aminifar, Lund University

Examiner:
Maria Kihl, Lund University

July 5, 2023

Karolinska Institutet

LUNDS UNIVERSITET

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

In biology, cell counting provides a fundamental metric for live-cell experiments. Unfortunately, most researchers are constrained to using tedious and invasive methods for counting cells. Automatic identification of cells in microscopy images would therefore be a valuable tool for such researchers. In recent years, deep learning-based image segmentation methods such as the U-Net have been explored for this task. However, deep learning models often require large amounts of labeled data for training. For identifying cells in microscopy images, this type of labeled data is commonly generated through manual pixel-wise annotations of hundreds of cells. To address this problem, we explore an approach for automatically generating large numbers of labeled examples by imaging cells that were stained with a fluorescent dye. By using fluorescence microscopy alongside non-invasive microscopy, we obtain visualizations of the positions of nuclei in each cell image. We transform the fluorescence images into binary masks with a pipeline based on classical segmentation techniques: histogram equalization through CLAHE and thresholding using Otsu's method. We then use these masks as labels for the cell images, so that each image is accompanied by pixel-wise annotations of the nuclei. We generate datasets for three different cell types, and use them to train U-Net models for automatic cell identification. The trained models show excellent performance ($\sim 2\%$ false positives, $<1\%$ false negatives), on par with expert annotation. This method therefore shows great promise as a tool for biologists to perform automatic cell identification and counting. The trained U-Nets can potentially also be used for tracking cells in time-lapse imaging. These new data extraction methods could assist researchers in deepening their understanding of the phenomena that they are studying.

Popular Science Summary

Starving cancer cells of the amino acid *methionine* has proven to be a way of stopping them from growing and dividing. While details behind this phenomenon are blurry, researchers are investigating this so that it might one day be exploited to fight cancer. This is known as the *methionine dependency problem.*

A problem that many researchers in biomedicine face is that even though they can conduct cell experiments by giving cells some treatment (in this case, by removing their access to methionine), they do not have access to many methods for extracting experimental data. Researchers are usually interested in measuring cell counts in order to see how the number of cells changes over time when subject to their chosen treatment. Unfortunately, in order to count cells they often have to remove the cells from their nutrient solution and pass them through a measurement tool. This does not only take a lot of time (counting cells at a single time point can require hours of manual work) but can also have unexpected effects on the experiment. In addition, by exclusively focusing on cell counts, valuable information such as cell movements, divisions and deaths are lost. This information can be found by imaging cells through microscopy, but manually reviewing microscopy images is difficult and time-consuming. Furthermore, single experiments may yield tens of thousands of images and analyzing this copious amount of data manually is simply not feasible.

Recent development in artificial intelligence (AI) and machine learning has opened the doors to new ways of tackling this issue. AI techniques have been used to detect specific objects of interest in images for years, but some problems are more difficult than others. For example, while AI-based software in smartphones easily detects faces when using their cameras, identifying cells in microscopy images has proven to be a difficult task.

If we could train AI to detect cells in microscopy images, we could automatically analyze these tens of thousands of images in no time at all. If all cells were found, we could of course count them with ease. We could also track events such as movement, division and death, which would be very valuable for the researchers. There is only one problem with training AI: you need training data. Usually, lots of it.

For this kind of task, training data would need to consist of two things: cell microscopy images and pixel-wise locations of the cells in said images. These pixel-wise annotations are called *labels*. Because microscopy images of cells tend to be

hard to analyze even for humans, accurate labels are not easy to obtain. In addition, the images that specific research groups obtain might have unique properties because of experimental setups and chosen cell types. So, even if someone trained AI for this, it might not be easy to share it.

When working with small amounts of data, experts can generate labels by manually pointing out the locations of each cell. This process is difficult, time-consuming and becomes an enormous project for larger datasets. However, if there was an automatic way to do this, researchers could easily use their own data to train their own AI for cell counting and tracking purposes.

To solve this labeling problem, we set out to explore automatic labeling methods. We tried a new approach that involved a form of microscopy that separates cells from their background much more clearly. This technique is toxic to the cells and cannot be used for actual cell experiments, but it *could* be used together with non-toxic microscopy techniques to generate *pairs* of images. These pairs would consist of one normal, non-toxic cell microscopy image and one image of cells that were clearly separated from the background. When put together, one is a normal microscopy image of cells and one clearly shows the pixel-wise locations of the cells in said image. This is the kind of labeled data that you could use to train AI!

To start, we built a pipeline that automatically transformed the raw cell microscopy data into training data that could be used to train AI for cell identification. Then, with state-of-the-art deep learning techniques and our own automatically labeled data, we trained AI to identify cells. We applied our method to several "different-looking" cell types and obtained excellent results. Experts evaluated the work of the AI and found that the accuracy of the AI was at least on par with expert annotation. We then explored ways to couple the AI with existing cell tracking algorithms that have been proven to perform well as long as cell locations were provided. We ensured that the entire process would be fully automatic, so that researchers could make use of this software without having any experience in AI development.

With our method, it is our hope that we may assist all kinds of biologists in extracting valuable data from their cell experiments, and take us one step closer to solving the methionine dependency problem once and for all.

Acknowledgement

First and foremost, I want to give a huge thank you to Roland Nilsson, Ph.D who has been my primary supervisor at Karolinska Institutet and the source of this project. His welcoming approach combined with his genuine interest and drive has made working on this project a wonderful experience. He has provided me with assistance, inspiration and motivation during the entire process and one could truly not ask for a better supervisor. I also want to thank Mohamed El Husseiny, Ph.D who has been the key part in the biology-side of this project, providing me with the cell data (among other things) that has been fundamental for our work. Together with Roland, he taught me about the biological concepts that played parts in our work with enthusiasm. I also want to give my thanks to the other members in Roland's research group: Deniz Seçilmiş, Ph.D and Nina Grankvist, Ph.D, who alongside Roland and Husseiny welcomed me into their research group warmly and contributed with kind and inspiring presences in the workplace.

I am also very grateful to Maria Kihl, Professor and Amir Aminifar, Ph.D who made it possible for me to do this project by taking on the roles of examiner and supervisor respectively at Lund University. Amir also provided me with supervision throughout the project alongside Arthur Andreas Nijdam, one of his Ph.D students, who I also want to thank. Amir and Arthur both provided very helpful feedback for finalizing the thesis.

I want to give a huge thanks to Joakim Jaldén, Professor who played a fundamental part in this project by sharing his thoughts and ideas regarding cell identification and tracking with Roland and myself. His ideas are the foundations of our methodology and it is safe to say that this project would not be done without his guidance. Joakim also provided me with supervision during the project.

Finally, I thank my family and friends who have supported me during the entire process.

The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at Alvis partially funded by the Swedish Research Council through grant agreements no. 2022-06725 and no. 2018-05973.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Background

Methionine dependency is a metabolic phenomenon that has been found in cancer cells, where the cells cannot proliferate (grow and divide) without access to the amino acid methionine [2, 3]. Roland Nilsson's research group at Karolinska Institutet (KI) is studying this phenomenon, striving to understand the metabolic basis of it, so that it in the long term might be exploited to suppress cancer growth.

A common way to study this kind of phenomenon is through experiments on *cultured cells*. This refers to laboratory methods where cells are left to grow, move, divide and die under controlled circumstances in a nutrient solution. The cells that are analyzed in these experiments are usually from *cell lines*, which are cells that grow and divide without limit as long as nutrients are available. In order to see how the cells are affected by their experiments, researchers often try to track how the number of cells changes over time by counting them at specific time points. Unfortunately, counting cells is a difficult task and commonly available solutions are often inadequate. These methods are usually very time-consuming (potentially taking hours to count at each chosen time-point) and can impact the experiment in unexpected ways due to interference (e.g. temporarily removing the cells from the culture conditions in order to count them).

One approach for cell research is to analyze cells through imaging. One important data type for this approach is *time-lapse microscopy* images of cultured cells. This refers to sequences of images that are taken over time in order to monitor the cells. Time-lapse microscopy can reveal data such as cell counts, but also cell migration (movement), proliferation and how the cell-activity changes over time. However, these images can be hard to analyze even for humans, meaning that successfully extracting data from time-lapse microscopy images is a difficult and time-consuming task. To help analyze this data, a system using computer vision techniques could be used in order to identify individual cells in microscopy images, count them and track their actions. This would be helpful when studying cell activity over long periods of time. Such a system would not only automate the data extraction, but also potentially yield more accurate results by increasing the number of analyzed time points and removing human error as well as any unexpected effects that would come from interfering with the experiment to count cells.

In recent years, machine learning techniques (specifically deep learning techniques) have been growing increasingly popular for biomedical image analysis and several algorithms have been proposed for cell identification [4]. One such algorithm is the U-Net, a deep learning architecture that was built for image segmentation (separating parts of an image into objects of interest) in biomedical contexts [1]. A problem with using these techniques is that they often require large amounts of labeled data. In this context, labeled data would consist of cell images and pixel-wise annotations for each image, so that the pixels that are parts of cells are separated from the pixels that belong to the background. These annotations are called *labels* and are often made manually, which for cell images is very tedious, time consuming work and subject to human error [5]. In addition, if data is to be labeled manually, this labeling process needs to be repeated each time new data is introduced.

A microscopy technique called *fluorescence microscopy* can be used to make analyzing cell images significantly easier by amplifying the light intensity in specific parts of the cells, making them very easy to distinguish in a microscope [6]. Unfortunately, the process is toxic for the cells [7], which makes it undesirable for time-lapse experiments. It could, however, be used in conjunction with non-invasive microscopy techniques, such as phase-contrast [8], to generate image pairs: one microscopy image similar to what you would expect from a time-lapse experiment (here, phase-contrast) and one image where parts of cells (e.g. nuclei) are clearly displayed (fluorescence). These pairs could be compiled into training data, using phase-contrast images as input and processed fluorescence images as labels.

By programming algorithms for processing fluorescence microscopy images, one could build a pipeline for *automatically* generating training data suitable for training deep learning models (such as the U-Net), to identify cells in images that had been generated by non-invasive microscopy techniques used in time-lapse microscopy of cultured cells. Then, once the cells have been identified and segmented, one could apply established cell tracking algorithms [9] to the time-lapse images and extract "high-content" experiment data. This method could make it easy for research groups to train their own classifiers, using their own data, and use them to get more insight into how their experimental conditions affect cell migration and proliferation. Such insight would be helpful when studying methionine dependency as well as other cell phenomena.

1.2 Task

Our goal is to build an automatic pipeline that can be used to turn images from in-house microscopy into labeled training data and train U-Net models for cell identification in non-invasive microscopy images with this labeled training data. We then want to use the trained U-Net models to identify cells in time-lapse microscopy images so that existing cell tracking algorithms can be applied to the predictions. The high-level illustration of this is shown in Figure 1.1. To test the flexibility of our pipeline, we want to apply it to images of different cell lines (i.e different human cell types that have different appearances and properties).

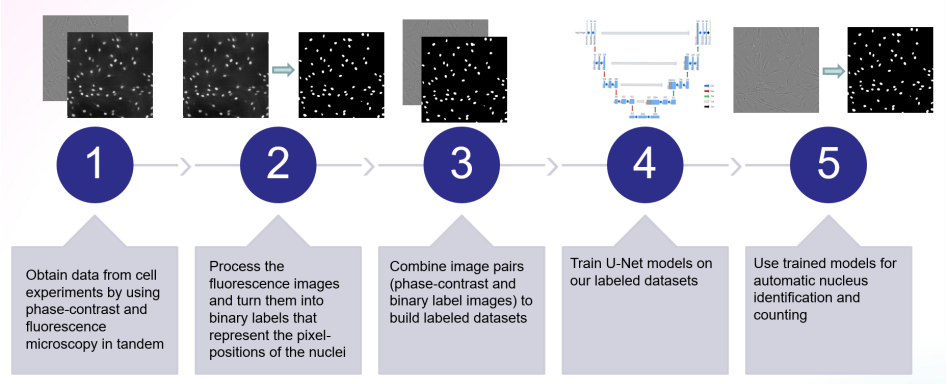

Figure 1.1: A high-level overview of the general idea of this project.

1.3 Purpose

The primary purpose for this project is to enable tools that can be used to assist in researching the methionine dependency phenomenon. However, these tools could be used for other research that feature cell experiments using time-lapse microscopy. With automatic cell identification from non-invasive microscopy images, researchers will not only get more accurate (and higher detail) results from their experiments, but also save a lot of time.

1.4 Limitations

Because of time constraints, we limit ourselves to the standard U-Net, since it has proven to be very reliable, even though newer variations of it have been proposed [10]. Likewise, we opt to using established settings (hyperparameters) for the U-Net.

For datasets, we limit ourselves to microscopy images from experiments featuring a small number of cell lines.

The thesis will focus on cell identification and not provide complete solutions for applications of it (e.g. cell-tracking). However, since it is of great interest, it will be briefly explored and some findings and ideas will be discussed in future work.

1.5 Thesis disposition

This thesis consists of eight chapters. Its general disposition is the following.

1. **Introduction**. This introductory chapter aims to outline the background of the project and state our goal as well as our limitations.
2. **Deep Learning Background**. As the first of two chapters about theoretical background, this chapter explains deep learning concepts that are relevant for understanding our methodology.

3. **Data Background: Microscopy and Image Analysis**. The second chapter about theoretical background gives some insight in the field of microscopy and states some relevant methods for processing microscopy image data.
4. **Software and External Resources**. As a prelude to the methodology, this chapter introduces the software and external resources that were used in our implementations.
5. **Methodology**. This chapter gives a detailed explanation of our chosen methodology, including our workflow, experimental specifics and evaluation methods.
6. **Results**. With a focus on evaluation, this chapter shows the results of the experiments that were introduced in the previous chapter.
7. **Discussion and Conclusion**. The penultimate chapter of the thesis aims to discuss our chosen methods and results. Some related projects are introduced and compared to ours.
8. **Future Work**. The final chapter of the thesis discusses some future work that can build upon what we achieved in this project.

I have not fetched any external images for the figures in this thesis except for a few logos from the external resources we have used. Cell images are taken from our own experiments and the figures have been made with platforms such as Lucidchart [11], the Python package Matplotlib [12] and Microsoft Excel.

Chapter 2

Deep Learning Background

In this chapter, I will explain background information behind the machine learning, and more specifically, deep learning that was used in this project. To begin with, I will discuss some background to the subject itself. Then, I will introduce the concepts and algorithms that were used in this project.

2.1 Machine Learning

Machine learning (ML) is a concept that is commonly associated with Artificial Intelligence (AI), and the terms are sometimes used interchangeably [13]. In reality, ML is simply a part of AI, and the concept of AI encompasses much more than just ML implementations [13].

ML revolves around building mathematical models that automatically adjust themselves by analysing data, in order to solve a given task. In other words, ML models "learn" to solve a problem by processing data. This can be done in various ways and can be applied to many different problems, but all revolve around programming a model and a training algorithm [14]. Commonly, ML is divided into three main branches: *supervised learning*, *unsupervised learning* and *reinforcement learning* [15]. This thesis will focus on supervised learning.

2.1.1 Supervised Learning

Imagine a task where you have some input data $\mathbf{x}$ that relates to some output data y. We don't know what this relation is and it might even be too complex to be stated explicitly. We do however have some data entries, where we know the outputs y from some given inputs $\mathbf{x}$. The idea of *supervised learning* is to build a mathematical model and "train" this model with this known data [14]. This is done by continuously telling the model what its output y_i should be for a given input $\mathbf{x}_i$, and letting the model adapt to fit this criteria. The data that is used for this process is called *training data* and consists of pairs $(\mathbf{x}_i, y_i)$, where y_i is called a *label* or *ground truth* and is the desired output of a given input $\mathbf{x}_i$, usually called a *feature vector* (often called *feature map* when its a 2-D matrix). The goal here is to train a model so that it can then *predict* what the label should be from any given input $\mathbf{x}$, including values of $\mathbf{x}$ that the model has never seen before. This prediction is the output of the model and is often denoted $\hat{y}$ [14].

What really defines supervised learning is the use of *labels*, meaning that each input $\mathbf{x}_i$ is accompanied by a label y_i and that we "teach" the model to associate data with our labels [14]. The term "supervised" comes from this: that we, during the training, tell the model what its output y_i to a given input $\mathbf{x}_i$ should be, rather than letting the model group data on its own [14].

2.1.2 Training Data

The training data is often split into three parts of difference sizes: a *training set*, a *validation set* and a *test set* [16]. These subsets are all composed of random samples from the original dataset, but are mutually exclusive (no entry is present in more than one subset) [16].

The training set is usually the largest subset and is commonly comprised of 60-80% of the total dataset. This is the data that is used for the training itself, i.e. the inputs $\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_n$ that are fed to the model during training and the corresponding labels $y_0, y_1, ..., y_n$ [16].

The validation set consists of data that the ML model can be tested on during training and serves as a means for evaluating the model on new data inside the training process [16]. In some cases, the validation set is used to tune the pre-defined settings of the model by exploring different model configurations and checking how well the different models perform on new data.

The test set is made of data that is not shown to the ML model during the training at all. This can be used to see how the final ML model performs on new data. Thus, its primary use is evaluation [16]. Note that good performance on a test set does not guarantee good performance on *other* new data. The test set is not used during training, but since people often retrain their models using different configurations in order to find models that perform better on the test set, models will eventually be somewhat biased towards the used test set. To avoid this bias, a test set should ideally only be used once.

2.1.3 Output Space: Numerical or Categorical

Depending on the task you are trying to solve, models trained through supervised learning might have very different output spaces [14]. A model trained to predict humidity based on some weather features might have a numerical output space consisting of an infinite number of potential labels (e.g. $\{y \in \mathbb{R} : 0 < y < 100)\}$, whereas a model trained to recognize a certain set of fruit might have a categorical output space consisting of a finite number of potential labels (e.g. $y \in \{\texttt{Apple, Banana}\}$). One could also simply have a binary problem, such detecting fraud based on certain features, where the output space would simply consist of the binary labels $y \in \{\texttt{True, False}\}$ or simply $y \in \{0, 1\}$. The output space defines what kind of problem the ML model is trained to solve: if the output space is numerical, the problem is a regression problem, but if the output space is categorical, the problem is a classification problem. If the ML model is trained to solve a classification problem, it is called a *classifier* [14].

2.1.4 Classifiers

The classifier is a common concept in ML where a trained model groups different inputs into specific classes [14]. When built using the supervised learning approach, the classes will consist of the predefined labels from the training data. Briefly, a classifier that is trained using supervised learning is built to classify inputs $\mathbf{x}_i$ into classes y_i, where $\mathbf{x}_i$ is a feature vector and y_i is the corresponding label. This concept is displayed in Figure 2.1 [14].

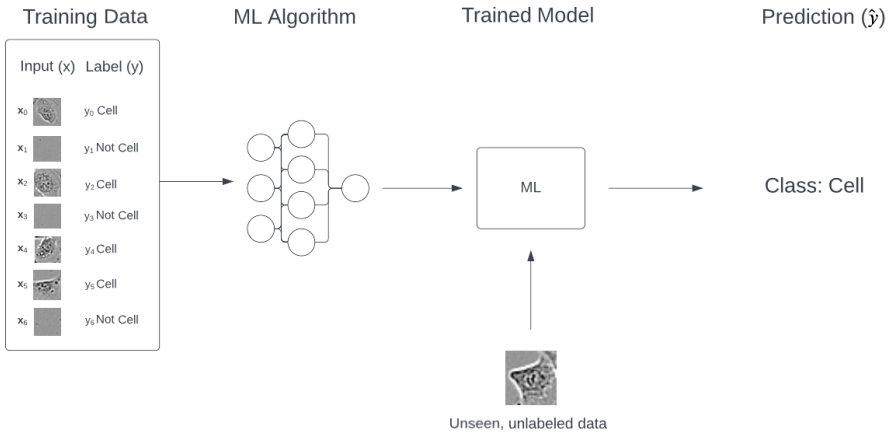

Figure 2.1: A classifier, trained using supervised learning to classify images into the classes "Cell" or "Not Cell". The figure displays the workflow of training a model and using it to predict the class of new, unlabeled data.

2.1.5 Image Segmentation using Machine Learning

One task that can be performed using a ML based classifier is *image segmentation*, a technique for identifying specific objects or areas of interest within an image [17]. This can be done by classifying the pixels in an image X_i, giving each pixel $X_{i_{j,k}}$ a label $Y_{i_{j,k}}$, where X_i is the i:th image in the dataset and (j, k) are pixel coordinates, thereby grouping the pixels into classes. This can be used with a large output space in order to find and classify objects of different classes in a single image, or if the goal is to find objects of a single class, the task can be reduced to a binary problem. Then, each pixel $X_{i_{j,k}}$ can simply be classified as "object of interest" or "not object of interest" (often just referred to as background). This means that the values in Y will be $Y_{i_{j,k}} \in \{0, 1\} \forall (i, j, k)$ and the label matrix Y_i for a given input image X_i will be represented as a binary image. This binary image is an image that exclusively consists of two possible pixel-values (usually 0 and 1, black and white), where each pixel-value represents one of the two classes [17].

The process of classifying all pixels in an image that belong to the same type

of object to the same class is called *semantic segmentation* [18]. An example of a binary classification problem in a semantic segmentation context is displayed in Figure 2.2.

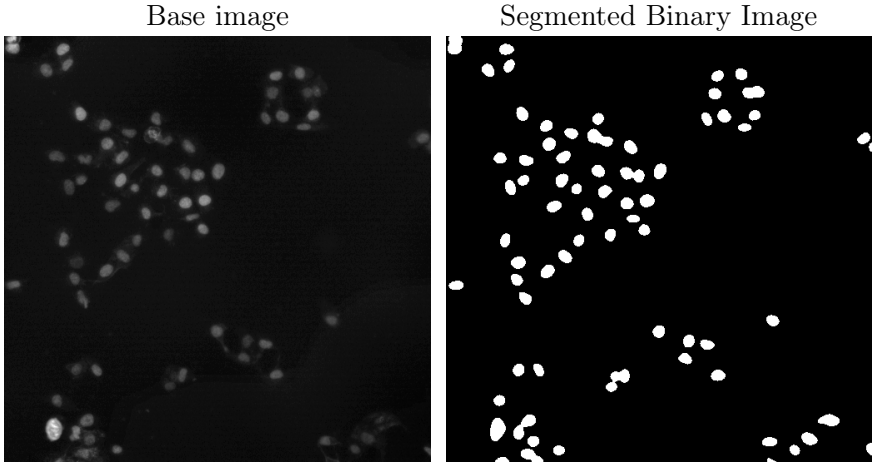

Figure 2.2: A binary segmentation example. Here, the task is to identify nuclei (the lighter, slightly oval objects) and separate them from the background. To the left is the input image and to the right is a segmented output in the shape of a binary image where white represents nuclei and black represents background.

2.1.6 Deep Learning

Image segmentation has recently been improved upon by the use of *deep learning*: a group of ML methods that use ML models called *artificial neural networks*. Deep learning models are flexible and highly customizable and can be applied to solve a range of tasks, such as language translation, identifying diseases, object recognition [19] and of course, image segmentation [18]. Semantic segmentation through deep learning is one of the main focal points of this report.

2.2 Deep Learning and Artificial Neural Networks (ANN)

2.2.1 What is an Artificial Neural Network?

Artificial neural networks (ANN) are mathematical models that are inspired by biological neural networks such as those composing the human brain [20]. They can be seen as AI methods that aim to reproduce human intelligence by imitating the way we think [21].

The ANN concept is not new. In fact, similar models were being conceived in the 1940s [22]. Unfortunately, the first models significantly predated the modern computers and research in the field eventually came to a standstill after an ANN-criticizing book was published, claiming that ANNs would probably not develop

into anything useful [21].

Later, when computers became more available, progress in the field of ANNs started again. In the early stages however, scientists were still held back by the low computing power of early-stage computers. Simple ANNs saw much use in the early 1990s and in the last 15 years, tremendous progress has been made now that the available computing power is on a whole different scale [21] [23].

Today, ANN based models are widely used [23], but even with today's technology, some very complex models can require hundreds or even thousands of high-end GPUs to be trained within a feasible time frame, which might still take several weeks or even months [24]. ANNs, just like other ML models, need to be trained and learn from data [19].

2.2.2 The Perceptron

The perceptron is the simplest ANN and serves as a good starting point for understanding how these models work. It is displayed in Figure 2.3. It consists of a *neuron*, also sometimes referred to as a *node* or a *unit*, which takes some input feature vector $\mathbf{x}$ $(x_0, x_1, ..., x_n)$ and gives some output $\hat{y}$. Each input x_i is multiplied by a *weight* w_i. In addition to this input, a bias b is added. If the number of input features is n, the information that the perceptron obtains can be written as [16]:

$$h(x) = \sum_{i=0}^{n} x_i w_i + b \tag{2.1}$$

The result is passed through an *activation function*. The output from this activation function becomes the output of the perceptron: $\hat{y}$. This activation function could, for example, be the unit step function:

$$f(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

Combining the input with activation function, we can see the function that the perceptron embodies:

$$\hat{y} = \begin{cases} 1, & \text{if } \sum_{i=0}^{n} x_i w_i + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

In this case, the perceptron outputs either 1 or 0 depending on the output. This can be seen as a binary classifier, classifying different inputs into the "1" class or the "0" class. How it will do this depends on the weights and bias. The input is generally unknown, but by tuning the weights and bias, you would change the behaviour of this classifier. However, the perceptron is very simple and cannot perform advanced classification tasks. Specifically, it can only complete this task if the two different classes are linearly separable [16].

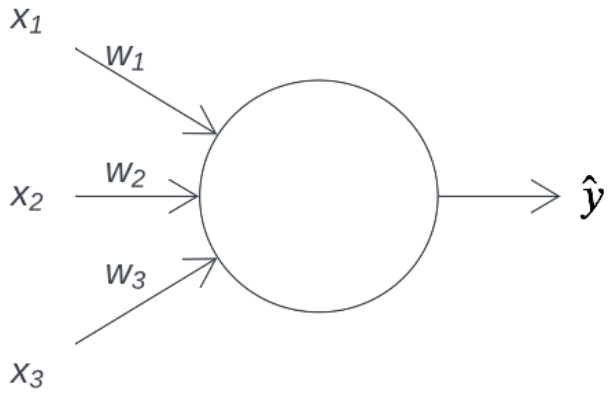

Figure 2.3: The perceptron, showing inputs, weights, the neuron/node itself and the output.

2.2.3 The Multi-Layer Perceptron (MLP)

The multi-layer perceptron (MLP) takes the perceptron concept to the next level and essentially builds more complex ANNs by building *layers* of perceptrons [16]. In the MLP, the input vector is fed into each node in the first layer, each feature getting multiplied with a weight. The inputs get summed individually in each perceptron and passed through their activation functions. The outputs from the first layer of perceptrons then become the inputs to the nodes in the second layer of perceptrons, and so on, until the final layer (called the *output layer*) is reached. Because of this behaviour, it is called a *feed-forward network*. The layers between the input layer and the output layer are called *hidden layers*, since we don't explicitly know the inputs and outputs to the nodes in those layers. The MLP concept is displayed in Figure 2.4 [16].

For binary classification, the MLP's output layer consists of a single output node. Rather than having a binary output space, the output is usually a probability p. The probability p is the predicted probability that the input $\mathbf{x}_i$ belongs to the class $y = 1$, and it follows that $(1 - p)$ is the probability that the same input belongs to the class $y = 0$. When reading this output as a classification result, the class with the higher probability is read as the predicted class. The actual classification then becomes [16]:

$$\hat{y} = \begin{cases} 1, & \text{if } p > 0.5 \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

Thus the output not only gives us a prediction, but also a metric of the MLP's confidence. If, for example $\hat{y}_i = y_i = 1$ and $p = 0.99$, we know that $\mathbf{x}_i$ was classified correctly and that the MLP was very certain in this classification, which is what we strive for. On the other hand, if $\hat{y}_i = 1$, $y_i = 0$ and $p = 0.99$, we find that $\mathbf{x}_i$ was incorrectly classified and that the model did so with great certainty, which is a problem. Finally, if p is close to 0.5, we see that the MLP is not sure how to

classify $\mathbf{x}_i$.

The activation functions play a fundamental part in what makes the MLP a viable model. By applying functions that apply non-linearity to the network, the MLP can adapt to recognize complex patterns. More thorough information regarding activation functions as well as explanations regarding the functions that were used in this project will be explained in Section 2.4.4.

An understanding of MLPs makes understanding other deep learning models a lot easier. In the following sections, I will explain how MLPs are built and used, so that I can then explain how deep learning was used in this project.

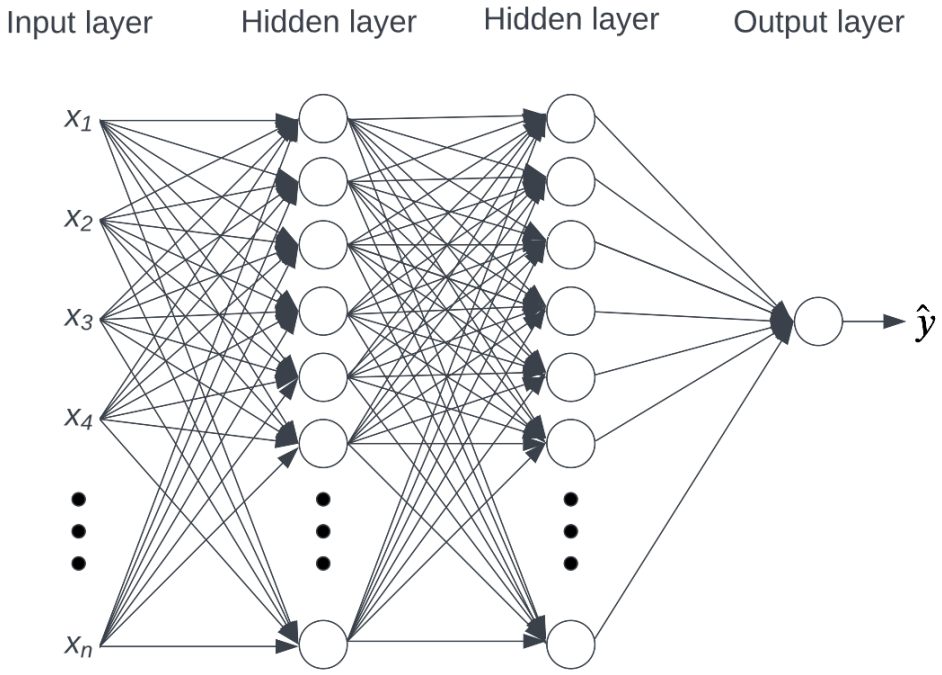

Figure 2.4: An MLP with some number of input nodes, two hidden layers and one output node.

2.2.4 Training

In order for the MLP (or any ML model for that matter) to work, it needs to be *trained* on data. The training process needs to be programmed, but the training itself is fully automatic. MLPs are trained using supervised learning and a common training process may look like this [14]:

1. Feed the MLP the data in the training set
2. Find how much the MLP's output differs from the ground truth (the known labels, the desired output)
3. Change the value of the weights to some combination that would yield an output that is closer to the ground truth (for this input)

4. Test the MLP on the validation set
5. Record the MLP's performance on the training set as well as the validation set
6. Repeat the entire process until some maximum number of iterations have been run or until the MLP stops improving

This amounts to an optimization problem of finding the optimal combination of weights for maximum performance.

2.2.5 Weights and Biases: Trainable Parameters

The weights and biases, along with the architecture of the network, define the MLP's behaviour [16]. Regardless of how well a specific architecture suits a task, these parameters need to be tuned for the MLP to be able to successfully solve classification tasks. It is entirely possible that a certain combination of parameters can solve a task within a certain dataset quite well, but performs worse on another dataset within the same task [16].

2.2.6 The Loss Function

The *loss function*, also known as the *cost function*, gives a value based on two factors: how often the model correctly classifies inputs and how "confident" it is [16]. Correct classifications with high confidence lead to a low loss, while incorrect and "insecure" (p not close to 0 or 1) classifications increase the loss.

The loss function L is a function of the ground truth y and the prediction $h(x) \approx \hat{y}$, where $h(x)$ is the function that the model embodies. The prediction $\hat{y}$ is thus a function of the inputs as well as all weights and biases of the MLP. This function may, however, be very complex.

The goal of training an MLP is usually to tune the parameters to minimize the loss function on the validation data, and by doing so, maximizing the performance on new data [14]. The loss functions that were relevant to this project will be discussed in Section 2.6.

The process of feeding data to the MLP and evaluating its output based on loss is called *forward propagation*. The process of computing how to tune the parameters based on the results of the forward propagation is done through an algorithm called *backpropagation* [14].

2.2.7 Backpropagation

Backpropagation is an algorithm that seeks to compute the *gradient* of the loss function ∇L for some input $\mathbf{x}_i$ and its corresponding label y_i [16]. The following is a rough overview of the algorithm. For a more detailed explanation, see [14].

When computing the gradient $\nabla L(h(\mathbf{x}_i), y_i)$, we quickly find that it depends on the output from the output node of the network as well as the provided label. Thus, to compute the gradient $\nabla L(h(\mathbf{x}_i), y_i)$, we need to compute the partial derivatives $\frac{\partial L}{\partial w_{l,j,k}}$ and $\frac{\partial L}{\partial b_l}$ for some input $\mathbf{x}_i$, where $w_{j,k,l}$ the weight between node k in layer $l-1$ and node j in layer l (here, the output layer) and b_l is the bias input

to layer l. These partial derivatives in turn depend on the weights and biases of the previous layer, which in turn depend on the weights and biases in the layer before that and so on until we reach the input nodes [16].

In order to compute $\nabla L(h(\mathbf{x}_i), y_i)$ for some input $\mathbf{x}_i$ we therefore need to recursively compute the partial derivatives $\frac{\partial L}{\partial w_{l,j,k}}$ and $\frac{\partial L}{\partial b_l}$ for all weights and biases in the network which is done using the chain rule. The gradient $\nabla L(h(\mathbf{x}_i), y_i)$ is then used for minimizing the loss function with an optimization algorithm called *gradient descent* [16].

2.2.8 Gradient Descent

Since MLPs often consist of millions of parameters [16], the loss function becomes a function that depends on millions of variables. Minimizing it becomes a difficult optimization problem and computing the global minimum cannot be solved in closed form [14]. However, if one could compute the direction in the multi-dimensional space of the loss function, in which the function's value decreases, one could take a step in that direction. By repeating this, one would eventually arrive at some local minimum. This is the idea behind *gradient descent* [16].

The direction of the *gradient* ∇f of a function f at a point P_n is the direction in which f has its greatest increase from the point P_n. It follows that the direction opposite of the direction of ∇f is where f has its greatest decrease from the point P_n. This direction is simply the direction of the negative gradient $-\nabla f$.

When $-\nabla f$ is known, one can "take a step" of size η in its direction. Then, one arrives at a new point P_{n+1} where $-\nabla f$ needs to be computed again for this new point P_{n+1}. This becomes an iterative stepping algorithm where the steps can be written as:

$$P_{n+1} = P_n - \eta \nabla f(P_n) \tag{2.5}$$

Given that the steps are sufficiently small, one will eventually converge at a local minimum. If the steps are too large, it is possible to diverge instead, since you may follow a certain direction for far longer than you should. The algorithm is usually set to terminate when the steps become very small (when $P_{n+1} - P_n < \epsilon$ where ϵ is some tolerance) or when some chosen maximum number of iterations have transpired, depending on implementations [16].

In a large MLP, you have many weights (in the scale of thousands or millions) and you therefore have a loss function L existing in a space consisting of many dimensions. It is therefore hard to fathom or illustrate what this algorithm would look like in an MLP context, but in a three-dimensional space, one could imagine following some slope down into a valley, like illustrated in Figure 2.5.

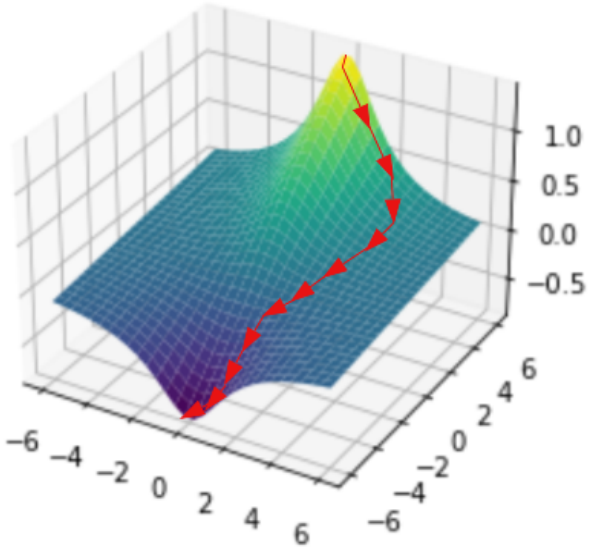

Figure 2.5: Gradient descent, roughly illustrated in three-dimensional space. After starting at some point, one continuously takes steps down the steepest slope, until some local minimum is reached.

2.2.9 Backpropagation with Gradient Descent

When feeding the MLP some input data, the network is in a state θ_n where the weights and biases have some values. After forward propagation, the loss function gives us some value $L(h(\mathbf{x}_i), y_i)$, where y_i is the known label attached to the input $\mathbf{x}_i$. Gradient descent is applied here by computing the negative gradient $-\nabla L$ (the direction in which the loss function decreases the fastest) through backpropagation and taking a step in that direction. After taking a step, one arrives at a new point θ_{n+1}, which is a new network state with some new combination of weights and biases. To get there, the weights and biases of the network needs to change. This update rule, with respect to the loss function L and the set of weights and biases θ_n, becomes:

$$\theta_{n+1} = \theta_n - \eta \nabla L \tag{2.6}$$

By following this gradient descent algorithm and minimizing the loss function for different data, one may improve the performance of the MLP. Gradient descent based algorithms are by far the most common approaches to optimizing ANNs. In deep learning, there are three common approaches to gradient descent [25]:

- Stochastic gradient descent (SGD): Compute the gradient based on a randomly chosen pair $(\mathbf{x}_i, y_i)$ in the training set. Then, update the weights. Since this updates the weights based on a single pair $(\mathbf{x}_i, y_i)$, it makes each

step faster but might lead to noisy steps, especially if the training set has great variation.

An alternate approach to SGD is to select these pairs $(\mathbf{x}_i, y_i)$ by iterating through the training set instead of randomly selecting one pair $(\mathbf{x}_i, y_i)$ at a time. In either case, with ∇_θ being the gradient based on the current parameters θ, the update rule is [25]:

$$\theta_{n+1} = \theta_n - \eta \nabla_\theta L(h(\mathbf{x}_i), y_i) \tag{2.7}$$

- Batch gradient descent: process the entire training set, compute the gradient and update the weights. This approach takes "smoother" steps than SGD since the final gradient is based on the entire training set, but takes longer and converges slower, especially for large training sets. Also, rare patterns in the training data might be overlooked by mostly basing the final gradient on common patterns. If N is the size of the training set, this update rule becomes [25, 26]:

$$\theta_{n+1} = \theta_n - \eta \frac{1}{N} \nabla_\theta \sum_{i=1}^{N} L(h(\mathbf{x}_i), y_i) \tag{2.8}$$

- Mini-batch gradient descent: same as batch gradient descent but process small parts called *mini-batches* instead of the entire training set before updating the weights [25, 26]. The size of these mini-batches is called the *batch-size* m. This serves a middle ground between SGD and batch gradient descent, compromising between speed and precision. If $m = 1$, this is the same as the second version of SGD [25, 26].

$$\theta_{n+1} = \theta_n - \eta \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(h(\mathbf{x}_i), y_i) \tag{2.9}$$

Since gradient descent gets stuck in critical points (points where $\nabla_\theta L = 0$), local minima and saddle points both serve as obstacles in the search of the global minimum. That said, studies have shown that getting stuck in saddle points is not a common phenomenon [27]. Saddle points can, however, slow the algorithm down significantly [28]. Still, a fundamental problem with gradient descent is that even if it can pass by saddle points, it still only converges to a local minimum. Several optimization algorithms have been proposed in attempts to tackle this problem (as well as the performance issues from saddle points), some of which are described in a Section 2.5.5.

2.2.10 Batch Normalization

When using some version of mini-batch gradient descent, it has been proven useful to apply an algorithm called *batch normalization* to the inputs to hidden layers [29]. The goal of this algorithm is to speed up and stabilize training by mitigating the effects of a phenomenon referred to as the *internal covariate shift*. This phenomenon is described as the constantly changing distributions of each layer's

inputs during training which stem from the parameter changes in previous layers. This makes training slower and more complicated, requiring low learning rates (η) and careful parameter initialization [29].

To account for the internal covariate shift, batch normalization is done by re-scaling the input mini-batch $B = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m\}$ of size m to each layer based on the mean μ_B and variance σ_B^2 in B [29]:

$$\begin{aligned} \mu_B &= \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_i \\ \sigma_B^2 &= \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_i - \mu_B)^2 \end{aligned} \tag{2.10}$$

Since the elements in B are vectors of length n, the variables $\{x_{i_1}, x_{i_2}, ..., x_{i_n}\} \in \mathbf{x}_i$ are normalized separately. A small value ϵ is added to the denominator to avoid division by 0 [29]:

$$\hat{x}_{i_j} = \frac{x_{i_j} - \mu_{B_j}}{\sqrt{\sigma_{B_j}^2 + \epsilon}} \tag{2.11}$$

To avoid changing what the layer itself may represent, two trainable parameter vectors γ and β are introduced, so that the result of the batch normalization transform becomes [29]:

$$BN_{\gamma_j,\beta_j}(x_{i_j}) = \gamma_j \hat{x}_{i_j} + \beta_j \tag{2.12}$$

When using batch normalization, the output of the BN transform will be the input to the following hidden layer. This algorithm has also proven to act as a form of *regularization* [29], which is the topic of the following section.

2.2.11 Overtraining and Regularization

When training an MLP on some training data, it is possible to achieve fantastic results on that specific dataset, to the extent of even catching weird outliers with 100% accuracy [16]. If this happens, there is a chance that the network is *overfit* (also known as *overtrained*) on this dataset. What this means is that the network has been tailored to achieve good results on the specific samples of data, and will therefore probably not be able to achieve good results on new data.

Overfitting is tightly linked to the *model capacity*, i.e. the complexity of the model. If the model is too complex for the data its being trained on, it might be able to "memorize" the patterns in the data and adapt fully to it. Then, overfitting occurs.

To counter overfitting by restricting the model capacity, *regularization* techniques are used. Regularization is the process of making a function simpler. Regularization may lead to worse performance on the samples in the training data, but will ensure that the network can handle new data within the same task to some extent [16]. The concept of overfitting and regularization is shown in Figure 2.6.

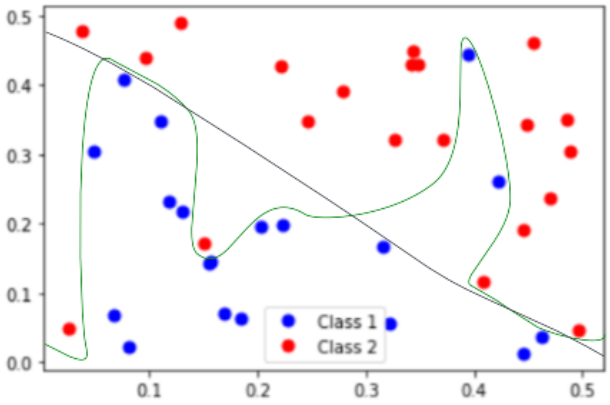

Figure 2.6: A typical classification problem: tune a function that can separate one class (blue dots) from the other (red dots). One function (black line) is a simpler, regularized function that does not separate perfectly for this data, but if other distributions are similar, it will still do quite well on new data. The other function (green line) fits everything perfectly on this data, including outliers. This function is very complex and evidently tailored to succeed on this particular data and is likely to get worse results on new data. This function is overfit on this data.

Overfitting can be detected in ANNs. Typically, the performance on the training set will increase, while the performance on the validation set decreases [14]. What this means is that the network is getting better at classifying the data in the training set while simultaneously getting worse at classifying new data. If this occurs continuously, one usually wants to stop training early. This technique is called *early stopping* [14].

Another common regularization technique is called *dropout* and acts by directly lowering the model capacity during training by randomly ignoring parts of the network. This technique will be explained in greater detail in Section 2.4.3.

Although a training that is based on validation loss can avoid overfitting on the training set, the problem of overfitting extends further. By optimizing the model to find the lowest validation loss you will build a model that performs well on that particular data. This can even be extended to the test set: if you keep re-training a model, using different configurations, in attempts to improve performance on the test set, the model may end up overfit on this data too.

It can be difficult to know the extents of overfitting and although several regularization techniques exist, it is an important concept to keep in mind while training and testing any ML model. The effects of overfitting and regularization are studied extensively and exact details of how model capacity can relate to overfitting is not perfectly clear [30].

2.3 ANN Architecture and Hyperparameters

MLPs (and ANNs in general) are defined by their *architectures* and training algorithms [14]. The architecture refers to how the model is built, such as the number of hidden layers that are used, the number of nodes in each layer and which activation functions they use. The training algorithm has several variables, such as choice of optimization algorithm, the chosen loss function and the number of passes through the training data. The variables that define the architecture as well as the training algorithm are called *hyperparameters* and can be split into two catagories: model hyperparameters (architecture) and algorithm hyperparameters (training algorithm) [14].

An important distinction is that "parameters" usually refer to the trainable weights and biases of a network, while "hyperparameters" are specifics that are set by the programmer in advance [14].

2.4 ANN Model Hyperparameters

2.4.1 Hidden Layers and Nodes

The number of hidden layers as well as the number of nodes in each layer affects the behavior of the network [16]. With more hidden layers and an increased number of nodes in said layers, the network becomes more complex and *may* be more capable of learning complex patterns. A more complex network, however, also takes longer to train and is slower to use. It may also be more prone to overfitting [16].

There can theoretically be any number of hidden layers and any number of nodes in a hidden layer.

2.4.2 Weight Initialization

By initializing the weights with some values, one may affect how the gradient descent algorithm converges [31]. It is common to initialize weights with some uniform or normal distribution, but it is also possible to initialize the weights with some combination that is known to do well on a certain task [31].

For modern image segmentation models (that will be introduced in Section 2.9), authors recommend an initialization technique that takes the initial weights from a zero-mean Gaussian distribution with a standard deviation $\sigma = \sqrt{\frac{2}{N}}$, where N is the number of inputs to the node [1]. This technique was proposed in a paper by He et al. [32] and is therefore often referred to as *He normal initialization*.

2.4.3 Dropout

Dropout is a regularization technique that randomly disables some nodes (meaning that their inputs and outputs are temporarily ignored) in the network during training [33]. This makes it so that the network cannot get too dependent on some specific nodes or weights, resulting in a more generalized network. This is usually implemented by setting some probability for each node to disable itself [33].

Since the value of the dropout hyperparameter p is a probability, the value range is $0 \leq p < 1$, where 0 means that dropout is disabled and 1 would imply full dropout, i.e. no nodes are active and the network does nothing.

In MLPs it is common to apply a fairly high dropout probability, often around 0.5. In other architectures, such as the convolutional neural network that will be brought up in a later section (Section 2.7), it has been shown that a much lower dropout probability can be beneficial in some layers [34].

2.4.4 Activation Functions

Each node has an activation function assigned to it that is used to process its input [16]. The output from this function is then the output of the node. Various functions may be used and the chosen functions have a significant impact on the behaviour of the network. Commonly, the function that is assigned to the output nodes is not the same as the function assigned to the nodes in the hidden layers. Additionally, different nodes within the hidden layers may use different activation functions [16].

To keep this list short, it is limited to the activation functions used in this project. Note that there are many more that can be used, depending on the task and the desired structure of the network.

One common activation function for nodes in hidden layers is the *rectified linear unit* (ReLU) [14]:

$$f(x) = x^+ = \max(x, 0) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.13}$$

As shown in equation 2.13, this function simply passes its input x on, as long as x is a positive number. If it is negative, the function yields 0. This behaviour is illustrated in Figure 2.7. ReLU's primary purpose is to introduce non-linearity to the network. This is important if the network needs to solve non-linear problems. Also, without non-linearity between the layers, all layers in a network could be replaced by a single layer, which removes the point of having multiple layers completely [14].

A large benefit with ReLU is that it does not contribute to changing the magnitude of the gradient during backpropagation, since its derivative is always 1 or 0. For other functions, with derivatives $f'(x) < 1$, gradients may slowly shrink away during the chained multiplications of derivatives in the algorithm.

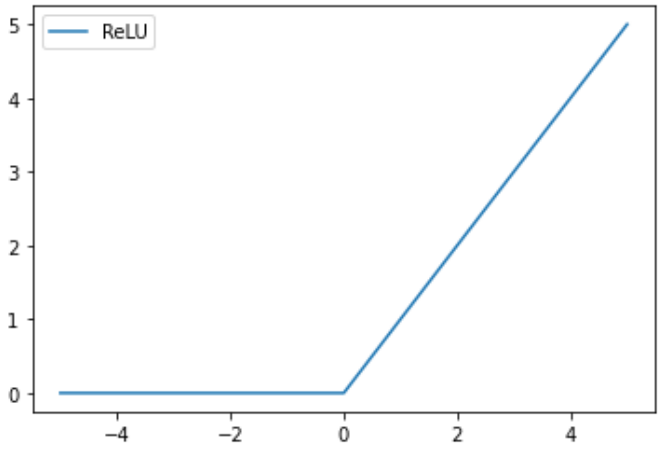

Figure 2.7: The rectified linear unit.

One interesting detail with ReLU is that it is not differentiable in $x = 0$. This could make it unusable for backpropagation, so in implementations, the derivative $f'(0)$ is arbitrarily set to $f'(0) = 0$ or $f'(0) = 1$.

For binary classification tasks (where only a single output node is needed), the output node commonly employs a *Logistic function*. In ML contexts, this is often just referred to as a *sigmoid* function [14]:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.14}$$

This function has a range of $0 \leq f(x) \leq 1$ and takes on the value $\frac{1}{2}$ when $x = 0$, forming the characteristic "S-shape" centered on $x = 0$, as displayed in Figure 2.8. The limitations given by the output range of the as logistic function and the fact that its differentiable makes it an ideal function for the output node, as it can can be read as a predicted probability which can be decoded to a predicted classification. In addition, unlike ReLU, its gradient is never zero. Zero-gradients are problematic since they do not give any information about how to improve the parameters [26].

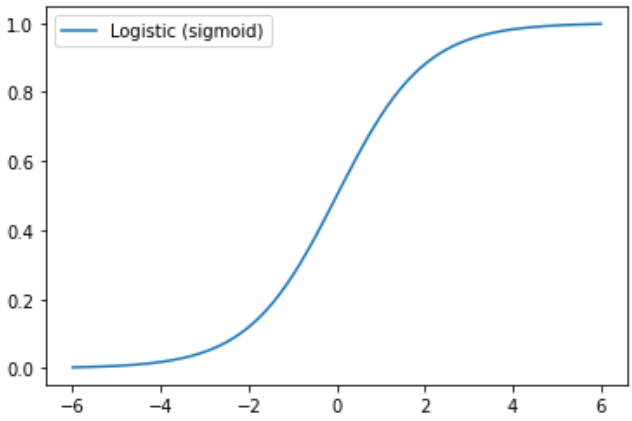

Figure 2.8: The logistic function.

2.5 ANN Algorithm Hyperparameters

2.5.1 Learning Rate

The learning rate is the size of the steps in the gradient descent η. A high learning rate means that the algorithm will converge faster, but risks oscillation and divergence. A low learning rate ensures convergence at the cost of a higher run-time, given that it doesn't terminate early [14].

Newer optimization algorithms may use changing learning rates. If such an algorithm is used, the hyperparameter simply denotes the starting point of the learning rate [25].

2.5.2 Number of Epochs

The *number of epochs* is the number of times the model processes the training set during the training. One *epoch* can be seen as one pass through the training set. In other words, after one epoch, the model has processed N data points $(\mathbf{x}_i, y_i)$ where N is the size of the training set. Typically, the model is tested on the validation set after each epoch.

It is important to note that the network may be updated multiple times during one epoch, i.e. one epoch does not mean one update. For example, if one trains a network using mini-batch gradient descent, one will have to do many updates before the entire training set has been processed.

The number of epochs can take on any value $N \in \mathbb{N}$. With a low value, the model might not become well trained, but a high value comes with the risk of overtraining. A high value can however be combined with early stopping to prevent this [16].

2.5.3 Patience

When using early stopping, one approach is to set a *patience* value. If, during the training, the validation loss does not decrease for a number of consecutive epochs equal to the patience value, the training will terminate [16]. After early stopping, one can revert back to the model with the lowest validation loss.

2.5.4 Batch Size

If using mini-batch gradient descent, the size of the batches needs to be specified. It is common for this hyperparameter to take some value 2^n where $n \in \mathbb{Z}^+$ [26]. The batch size might have upper limitations due to hardware specifications, such as memory.

2.5.5 Optimizers

As stated before, gradient descent comes with some issues, such as only converging to local minima and converging slowly. To improve on these flaws, several optimization algorithms have been proposed by different research teams [25]. These optimizers simply change the update rule of gradient descent into something else. The choice of optimizer is considered a hyperparameter [25].

Adaptive Gradients (AdaGrad)

When setting the learning rate η, it is hard to know if the chosen value is too low or too high without running the actual training. In addition, it might be beneficial to have different learning rates for different weights during the training [25]. With potentially millions of weights, it is not feasible to manually assign and test this. AdaGrad is designed to solve this problem [25].

AdaGrad is based on SGD but scales learning rates for each weight. This is done by keeping track of the past gradients and dividing the learning rate by the square root of the squared sum of past gradients. A small number ϵ is added to the squared sum of past gradients to avoid division by 0 [25]. The scaling makes it so that parameters with small partial derivatives receive a minor decrease to their learning rates while parameters with large partial derivatives get rapidly diminishing learning rates. AdaGrad's update rule is [25]:

$$\begin{aligned} \theta_{n+1} &= \theta_n - \frac{\eta}{\sqrt{s_n + \epsilon}} \nabla_\theta L \\ \text{where } s_n &= s_{n-1} + \nabla_\theta L^2 \end{aligned} \tag{2.15}$$

One flaw with AdaGrad is that the s_n is continuously increasing, meaning that the learning rate is continuously decreasing. After a while, the learning rates will be so small that all future steps are insignificant regardless if the algorithm has converged or not [25].

Momentum

Similarily to AdaGrad, the *momentum* algorithm uses information from past gradients in its update rule. The goal of momentum is not to adapt the learning rate to specific weights, but instead to avoid getting stuck in local minima by adding a moving average of previous gradients to each step. This moving average is computed by continuously adding the gradient to a sum of gradients that is multiplied with some decay $0 < \beta < 1$, making it so that the more recent gradients weigh more than older ones. The momentum update rule is [25]:

$$\begin{aligned} \theta_{n+1} &= \theta_n - s_n \\ \text{where } s_n &= \eta \nabla_\theta L + \beta s_{n-1} \end{aligned} \tag{2.16}$$

The decay parameter β is called the *momentum term* and is a new hyperparameter introduced with this algorithm [25]. Momentum helps with speeding up convergence and escaping local minima. In addition, it has been shown that momentum-based algorithms escape saddle points swiftly [35]. However, momentum still has oscillation issues: when approaching a minimum, the momentum algorithm will step back and forth which might still result in slow convergence [25].

Root Mean Square Propagation (RMSProp)

RMSProp is an extension of AdaGrad that uses a squared sum of past gradients s_n, but uses a moving average (like in momentum) instead of simply summing all previous squared gradients. This is to work around AdaGrad's problem with vanishing learning rates, while keeping its functionality of scaling the learning rate for different weights based on previous gradients. A decay parameter β is introduced, but is used slightly differently compared to momentum. Here, β is used to scale down s_n and $(1-\beta)$ is used to scale down the squared gradient. The RMSProp update rule is [25]:

$$\begin{aligned} \theta_{n+1} &= \theta_n - \frac{\eta}{\sqrt{s_n + \epsilon}} \\ \text{where } s_n &= \beta s_{n-1} + (1-\beta) \nabla_\theta L^2 \end{aligned} \tag{2.17}$$

Adaptive Moment Estimation (Adam)

Adam combines the benefits of momentum and RMSProp to get the best of both convergence rate and avoiding local minima. The name refers to using the first and second *moments* of the gradient. The algorithm uses moving averages that are estimations of the mean m (the first moment), and the uncentered variance v (the second moment) of the gradient. These are computed as [36]:

$$\begin{aligned} m_n &= \beta_1 m_{n-1} + (1-\beta_1) \nabla_\theta L \\ v_n &= \beta_2 v_{n-1} + (1-\beta_2) \nabla_\theta L^2 \end{aligned} \tag{2.18}$$

We recognize these as the moving sum of gradients and the squared moving sum of gradients from previous algorithms. Because β_1 and β_2 are often chosen to

be close to 1, the moments will be close to 0. This is referred to as being biased towards 0. To counteract this, the moments are "bias-corrected" [36]:

$$\begin{aligned} \hat{m}_n &= \frac{m_n}{1-\beta_1} \\ \hat{v}_n &= \frac{v_n}{1-\beta_2} \end{aligned} \tag{2.19}$$

The bias-corrected moments are then used in Adam's update rule, using the form of RMSProp where the learning rate is divided by the square root of the squared moving sum of gradients (with a small value ϵ added to avoid division by 0) and respect to the moving sum of gradients from momentum [36]:

$$\theta_{n+1} = \theta_n - \eta \frac{\hat{m}}{\sqrt{\hat{v}+\epsilon}} \tag{2.20}$$

Although there is no real consensus on which algorithm should be chosen [26], Adam is one of the most widely used optimizers and works well for most deep learning models [25].

2.6 Loss Functions

The loss function is at the core of the training process and the choice of loss function is an important hyperparameter. There are several popular ones and typically they work well for different tasks. In this project, we have explored *binary cross-entropy* and its extension *binary focal loss*. Note that these are only suitable for binary classification tasks, hence the names.

2.6.1 Binary Cross-Entropy

Binary cross-entropy, also known as *log-loss* due to its logarithmic nature, is a common loss function for binary classification tasks. This function compares the predicted probabilities of the output node to the known binary labels [14]:

$$L = -\frac{1}{N}\sum_{i=1}^{N}(y_i \cdot \log_2(p) + (1-y_i) \cdot \log_2(1-p)) \tag{2.21}$$

Here, y_i is the known label (ground truth) and p is the predicted probability (e.g. the result of the logistic function in an MLP output node for some input) of an input sample belonging to class $y = 1$. It follows that $(1-p)$ is the predicted probability of an input sample belonging to class $y = 0$.

By analysing this function, we can find that when $y_i = 1$, the inner function is:

$$-(1 \cdot \log_2(p) + (1-1) \cdot \log_2(1-p)) = -\log_2(p) \tag{2.22}$$

Similarly, when $y_i = 0$, the inner function is:

$$-(0 \cdot \log_2(p) + (1-0) \cdot \log_2(1-p)) = -\log_2(1-p) \tag{2.23}$$

So, for each sample, the loss is simply the binary logarithm of the predicted probability. The closer this is to the ground truth (1 for "class" or 0 for "not class"), the closer this will be to 0. Then, the value of the loss function is simply the average logarithmic prediction error in terms of probability. Thus we can see that minimizing this loss function will minimize the prediction errors, and so, the performance of the model will increase.

A potential issue with binary cross entropy is that each entry is treated the same. If there is a class imbalance in the training data, meaning that a certain class appears much more often than the other, one could get a pretty good value from the loss function by simply classifying everything as the common class. Another issue appears when the training set is large with many easy classifications (where the classification is correct and p is very close to 0 or 1). Then, these easy classifications may dominate the loss, making it so that the difficult classifications are not penalized enough and therefore never learned.

These are two very prominent problems in image segmentation, where background pixels may be very dominant in number and for the most part very easy to classify. If there are enough background pixels, the model may obtain a very low average loss by simply classifying everything as background. It will also not necessarily have to learn to solve the harder classifications. This has given rise to another loss function: binary focal cross-entropy.

2.6.2 Binary Focal Cross-Entropy

Binary focal cross entropy, also known as binary focal loss, is very similar to binary cross-entropy but puts a higher weight on difficult classifications and lowers the impact that the more common class has on the overall loss. The weighting of difficult classifications is done by introducing the modular hyperparameter γ [37]:

$$L = -\frac{1}{N}\sum_{i=1}^{N}((1-p)^{\gamma}y_i\log_2(p) + p^{\gamma}(1-y_i)\log_2(1-p)) \tag{2.24}$$

When $\gamma > 1$, the logarithmic factor gets downscaled tremendously when p is close to y_i. This is when classifications are easy, and the downscaling makes it so that they contribute little to the overall loss. In addition, this makes it so that there is very little to gain by increasing the confidence of easy classifications (e.g. being 99% certain instead of 95% certain), since the loss in these classifications will already be close to 0. Therefore, this behavior will not be striven for. This is to push the model into learning to solve harder classification tasks rather than perfecting easy classification tasks that it can already solve with great confidence [37].

Class weighting is done by introducing the α hyperparameter. It is simply a weight that can be set to a number $\alpha \in [0, 1]$ that reflects the class imbalance. Then, the entries of the common class will have less impact on the loss than the entries of the rarer class, making it so that accurate predictions on both classes are essential for a low loss. By including α for class weighting, we obtain the full binary focal cross-entropy function [37]:

$$L = -\frac{1}{N}\sum_{i=1}^{N}(\alpha(1-p)^{\gamma}y_i\log_2(p) + (1-\alpha)p^{\gamma}(1-y_i)\log_2(1-p)) \tag{2.25}$$

When using binary focal cross-entropy, the values α and γ are new hyper-parameters to be set before training. While α should be set depending on class representation in your dataset, the authors recommend the values $\alpha = 0.25$ and $\gamma = 2$ [37].

2.7 Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) are ANNs that are primarily used in image classification tasks [14]. These are defined by the use of *convolutional layers* and employ a different architecture compared to the fully connected networks (MLPs). A major problem with MLPs when it comes to image analysis is that the number of weights gets extremely high when the input is an image of high resolution. For example, an image of dimensions 1500x1000x1 results in a feature vector of size 1,500,000. In the case of an RGB image, this is then multiplied by 3 since each pixel has 3 color features, giving us a feature vector of size 4,500,000. In an MLP, each input has a connecting weight to each hidden node in the following hidden layer, meaning that for each node in the next hidden layer alone, there would be 4,500,000 weights. Because the number of weights in a layer is $N_l \cdot N_{l-1}$, where N is the number of nodes in layer l, the computational requirements become very high and the MLP architecture is typically not feasible for this kind of task [14].

Another problem for MLPs in image analysis is that their fully connected architecture fails to recognize patterns based on neighbouring pixels. We know that neighbouring pixels usually have more in common than two who are further apart, and this is information that we can use. If all pixels are handled independently, this information is lost [14]. Both of these issues are solved in CNNs by using their convolutional layers.

2.7.1 Image Channels

To understand convolutional layers, we first need to understand the concept of *image channels*. Channels can be seen as the third dimension of an image, meaning that in multi-channel images, each pixel is represented by a vector instead of a scalar [14]. The classic example is the 3-channeled RGB image, where each channel contains information regarding light intensities of a single color. In convolutional layers, the number of channels goes higher [14].

2.7.2 Convolutional Layer

A convolutional layer takes some input from a previous layer, performs some operation and then feeds some output to the following layer, just like hidden layers in MLPs. Instead of consisting of neurons, the convolutional layer consists of some number of *kernels*, sometimes referred to as *filters* [14].

The kernels are 3-dimensional tensors of some dimensions n x m x C (most commonly, $n = m$) where C is the incoming number of channels. Consider a simplified case where $C = 1$ and the kernels are reduced to 2-D matrices. These matrices "slide" over the input data, performing *convolutions* with n x m sized parts of the input, one at a time. The convolution is simply the sum of the element-wise multiplication between the input matrix I_l and the kernel K_p where I_l is a subset of the full image:

$$I_l * K_p = \sum_{j=1}^{m} \sum_{k=1}^{n} I_{l_{j,k}} K_{p_{j,k}} \tag{2.26}$$

The layer takes each value in the input data, transforms it into a weighted sum of itself and its surrounding values through the sliding window convolution, adds a bias and passes it on [14]. In an image, this means that the information of each pixel is treated based on the pixel itself and its surrounding pixels, which adds context to each operation. An illustration of a step in a convolutional layer with a single-channel input is shown in Figure 2.9 [14].

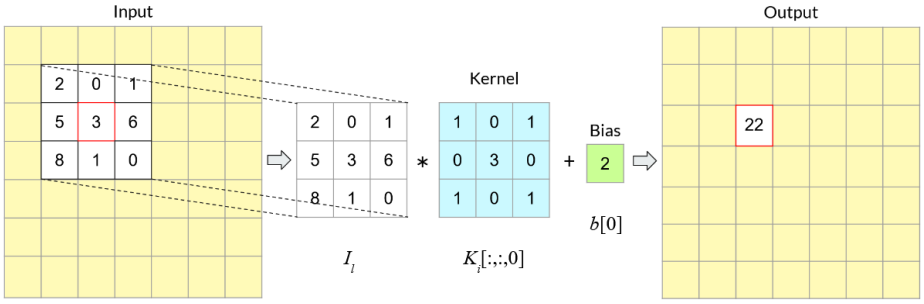

Figure 2.9: One step in a convolutional layer. Here, a part of the input surrounding a select pixel is convoluted with a 3x3 kernel, resulting in a single value output. This process is repeated until the kernel has processed the entire input data.

If the input consists of multiple channels, the kernels must be 3-dimensional with a depth equal to the number of incoming channels C so that the element-wise multiplication in the convolution can be performed. The output $O_{j,k}$ for a single step in the layer is then:

$$O_{j,k} = I_l * K_p + b_p = \sum_{i=1}^{C} \sum_{j=1}^{m} \sum_{k=1}^{n} I_{l_{j,k,i}} K_{p_{j,k,i}} + b_p \tag{2.27}$$

The convolutional layer consists of a number of kernels N. Each kernel processes the input independently and is accompanied by a bias, so that there is one bias for each kernel [14]. Each kernel yields its own output matrix based on the convolutions between itself and the input data as well as the added bias. This results in an N-channel output tensor. The result is processed by some activation

function, just like in the MLP. ReLU is often used, for the reasons explained in Section 2.4.4 [14].

The trainable weights of the convolutional layers are the values in the kernels and the biases [14]. This means that regardless of how large the input data is, the number of weights is constant (given that the input data doesn't increase in number of channels, which of course increases the depth of kernels and thereby the number of weights). In addition, the number of weights is usually very low compared to what an MLP would need. For example, if the kernels in a convolutional layer are of size 3x3 and 8 kernels are used, the total number of weights (excluding bias weights) is only $3 \cdot 3 \cdot 8 = 72$ per input channel. The low weight requirement and its way of adding context to all input pixels are the key strengths of the convolutional layer [14].

The way convolutional layers use a small set of weights to process small parts of the input data separately (as opposed to having a fully connected layer) is often referred to as *weight sharing*. The weight sharing in CNNs has some benefits (aside from the already established reduction of number of weights), a significant one being *translation invariance*, which is particularly useful when processing images. Since parts of the image are processed independently by the same kernels, it does not matter where certain patterns appear in the image. In addition, the low number of weights that come with the weight sharing may act as additional regularization.

2.7.3 Pooling Layer

When the input data has been processed in the convolutional layer, the output of the layer becomes a new matrix. The following layer (the one that takes the output from the convolutional layer as its input) is usually a *pooling layer*. The main purpose for this is to reduce the amount of computations needed in later layers of the CNN [14]. There are two common techniques for this: max-pooling and average-pooling.

Max-Pooling

Max-pooling is a simple algorithm that processes input by looking at some values at a time and passes on the highest number. This is done in a "sliding window"-manner, similar to the convolutional layer, but usually with a stride equal to the height or width of the window [14]. The larger this window (and thereby stride) is, the larger the down-scaling is. Max-pooling is shown in Figure 2.10.

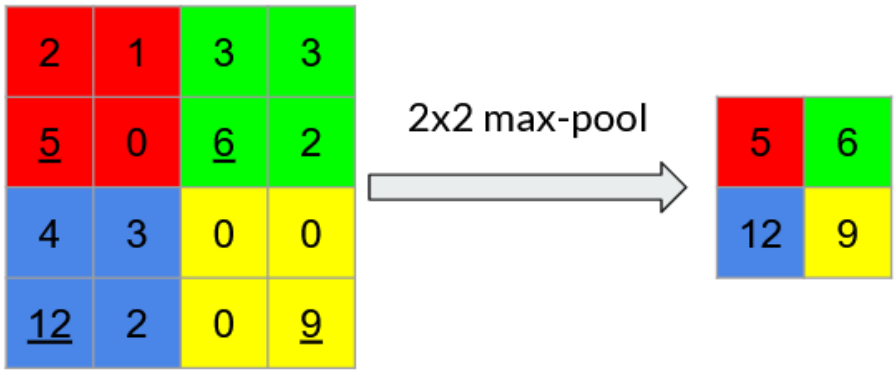

Figure 2.10: Max-pooling using a 2x2-sized window and stride = 2.

This emphasizes dominant and relevant features. Because of this, max-pooling is often the pooling operation of choice for high performance [38].

Average-Pooling

Average-pooling uses the same "sliding window"-manner as max-pooling, but it takes some average of the values in its window rather than just taking the maximum value, as the name suggests [14]. This can be done by computing the mean or the median.

2.7.4 CNN Architecture

Commonly, a CNN is built using a chain of convolutional blocks (a convolutional layer followed by a pooling layer) that eventually connects to a fully connected layer (also known as dense layer) which is then connected to a another fully connected layer or the output node(s). The feature map that connects to the fully connected layer usually needs to be flattened (reshaped form matrix to vector) first. Since the last part of the CNN is essentially an MLP, the output is the same.

As you go deeper into the network, the number of kernels in the convolutional layers usually increase exponentially, resulting in a rapidly increasing number of channels in the image data. This is why pooling is very important: without pooling, it would be very computationally expensive to use a CNN when a very high number of kernels would need to process a very large input. With pooling, the amount of processable data decreases exponentially, while the number of kernels increases exponentially, keeping the computational requirements in balance.

Conceptually, it is still quite similar to the MLP and most stated hyperparameters still apply. The main difference between the MLP and the CNN is that most hidden layers are convolutional layers instead of fully connected layers [14]. A rough illustration of a CNN is shown in Figure 2.11.

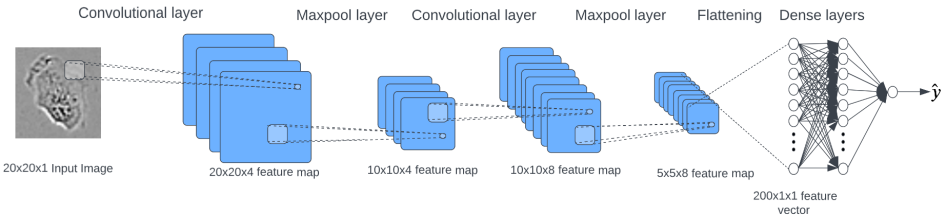

Figure 2.11: A CNN. The image data progressively shrinks in width and height, showing the down-sampling effect of the pooling. It simultaneously expands in depth, which is a visualization for the increasing number of channels.

2.7.5 The Kernels

To reiterate, the kernels consist of trainable parameters. Thus, the goal of training a CNN is to set the values in the kernels and the biases to some combination that minimizes the loss function when tested on the validation set. The training process here is the same as that of the MLP.

A trained CNN usually detects low-level features in the early layers such as edges and lines and higher level features in the later layers, such as entire objects [14]. This is a result of the parameters in the kernels, since the kernels control what the output of each layer will be.

2.7.6 Additional Hyperparameters in a CNN

In addition to the hyperparameters that are used in the MLP, some new hyperparameters need to be set in order to structure the new layers introduced in a CNN.

Kernel size

Kernel size, like the name suggests, controls the dimensionality of the kernels. It is most common to use odd, squared kernels (n x m, $n = m$, $n \in \{1, 3, 5, 7...\}$) but this is not a requirement [39]. These kernels have a useful property: being able to look at a center pixel and take some pixels surrounding it into account. The kernels may vary in size between layers. Using smaller kernel sizes makes for a lower number of trainable weights per kernel, which in turn might make a deeper, more complex network computationally feasible for some task, which might yield a higher performance than a simpler network with larger kernels [39].

Number of Kernels

Adding more kernels adds more weights to the CNN. This may increase the performance of the network at the cost of slower training and predictions.

Stride

The stride controls how many points the kernel will move with each step when processing input data [14]. With stride = 1, most pixels will be processed, only potentially missing some in the edges of the input. With a higher stride, the kernels will skip some pixels and the convolutional layer will serve as a down-sampling layer. The stride affects the kernels movement in all dimensions. This is displayed in Figure 2.12.

Padding

Padding can be used to prevent the edge pixels from being skipped, avoiding the small downsampling of the convolutional layer in the process. Padding is usually done by surrounding the input data with additional points of 0s, so that the kernel can "fit in" and process the input's edge pixels as center pixels in the kernel [14]. This too is displayed in Figure 2.12.

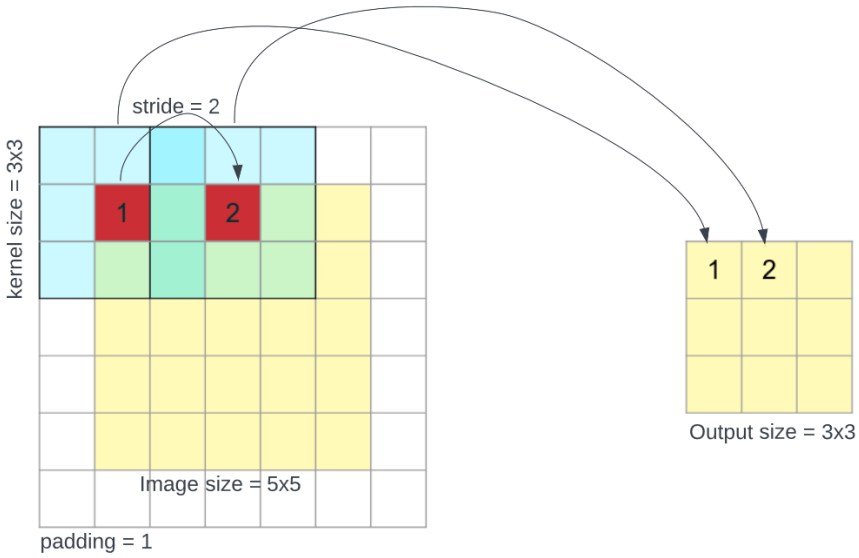

Figure 2.12: Input data being processed by a kernel with kernel size = 3x3, stride = 2 and padding = 1. Padding makes it so that the edges are processed and stride makes it so that many points are skipped, resulting in a down-sampled output. The numbers "1" and "2" in the input refer to the order that pixels are processed in (i.e. not the actual values). In the output, "1" refers to the result of the first convolution and "2" refers to the result of the second convolution.

Pooling Specifications

The choice of pooling technique in the pooling layers and the size of the pooling windows are hyperparameters that need to be set. A larger pooling window leads to more aggressive down-sampling. This can speed up the network but may lead to lower performance due to the loss of information in each layer.

2.8 Fully Convolutional Networks (FCN)

The fully convolutional network (FCN) is an architecture that obtained great results in semantic segmentation tasks [40].

2.8.1 FCN Architecture

The FCN is very similar to the CNN but does not end with fully connected layers [40]. Instead, the chain of convolutional blocks is followed by an up-sampling block (which is also built from convolutional layers) that returns the data to the shape of the original input. The output of the FCN is therefore very different from the CNN, though both are classifiers. Where the CNN outputs a predicted label for what the input image represents, the FCN instead outputs an image where each pixel is given a label, giving an image that outlines objects of interest instead of the original color [40].

The name of the FCN refers to the fact that it consists exclusively of convolutional layers (and their companions, the pooling layers). For more information regarding the FCN's background and properties, see the original paper [40]. An example of an FCN is shown in Figure 2.13.

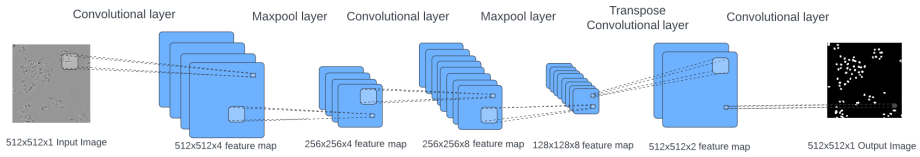

Figure 2.13: A rough illustration of an arbitrary FCN, using transpose convolution for up-sampling.

2.8.2 Up-Sampling with Transpose Convolution

There are several techniques for performing the up-sampling step [41]. One way is to apply a *transpose convolution*. This is an operation that is very similar to the convolution applied in convolutional layers. It takes an input and slides a kernel of some size, using some stride, applying the transpose convolution. Instead of taking the sum of the element-wise multiplications for the matrices, transpose convolution instead takes one value from the input, multiplies it with all values in the kernel and passes on all products to the output. Given a kernel that is larger than 1 x 1, the output matrix will be larger than the input, meaning that the input has been up-sampled. If a low stride is used, the matrix co-ordinates of some transpose

convolutions will overlap. These are simply added to each other, so that some parts of the output matrix is the sum of several transpose convolutions. With a higher stride, the output matrix becomes larger, resulting in a more aggressive up-sampling.

Transpose convolutional layers are built the same way as standard convolutional layers but apply this operation instead of the usual convolution. The kernels in these layers consist of trainable parameters, too. An example of a transpose convolutional layer is shown in Figure 2.14.

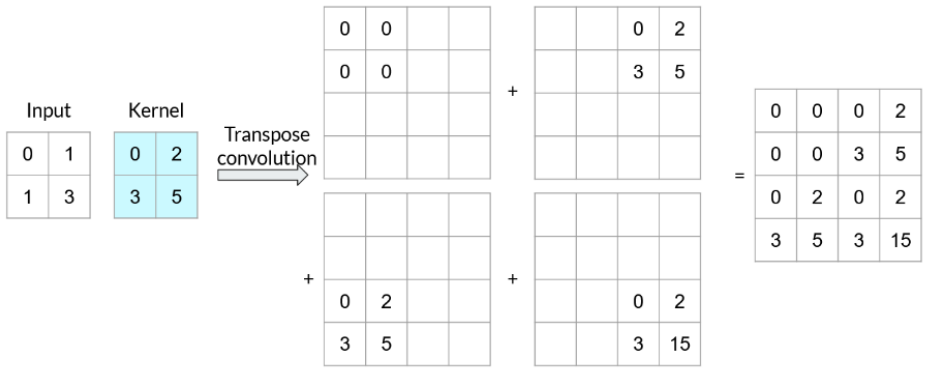

Figure 2.14: A transpose convolutional layer with an input feature map of size 2x2, using a 2x2 kernel. With stride = 1, the resulting output is of size 3x3.

2.9 The U-Net

The U-Net is an extension of the FCN that was proposed for image segmentation in biomedical contexts in 2015 [1]. Years later, it is still a very prominent architecture, being widely used in this field [42]. Reasons for this include generally good performance [42] as well as an ability to learn complex segmentation tasks from relatively small datasets [1]. Since labeled data is often hard to come by in biomedical contexts, this ability has proven to be especially useful. The U-Net served as the primary deep learning tool used in this project.

2.9.1 U-Net Architecture

The U-Net gets its name from its symmetrical U-shaped architecture. It is essentially an FCN using transpose convolution for up-sampling but with some special properties [1].

First, the down-sampling part consists of blocks of paired convolutional layers followed by a max-pooling layer [1]. The max-pooling is with a 2x2 kernel using stride = 2, making it so that the shape of the output of the convolutional block is

halved in both height and width. After each down-sampling step, the number of kernels in the following convolutional block is twice the number of kernels in the previous convolutional block [1].

At the end of the down-sampling, the up-sampling begins [1]. The up-sampling part also consists of convolutional blocks, but instead of being two chained convolutional layers followed by a max-pooling layer, these up-sampling blocks consist of a transpose convolutional layer for up-sampling, followed by two chained convolutional layers. In the up-sampling step, the number of kernels is halved, keeping the ratio between the feature maps and number of kernels consistent [1].

The number of up-sampling blocks is equal to the number of down-sampling blocks [1]. This symmetry yields an opportunity that is utilized in the U-Net. The feature maps from each down-sampling block (i.e the output of the second convolutional layer in the block) is saved before it is down-sampled through max-pooling and passed onto the next block. They are then concatenated to the transpose convolution in the corresponding up-sampling block (i.e. the transpose convolutional layer where the network up-samples the data to the resolution that is equal to the resolution that this convolutional block outputs) to give more context to the up-sampling and assist in reconstructing the image. In other words, the output of the first down-sampling block is added to the input of the last up-sampling block, the output of the second down-sampling block is added to the input of the second-to-last up-sampling block, and so on [1]. This architecture is shown in Figure 2.15.

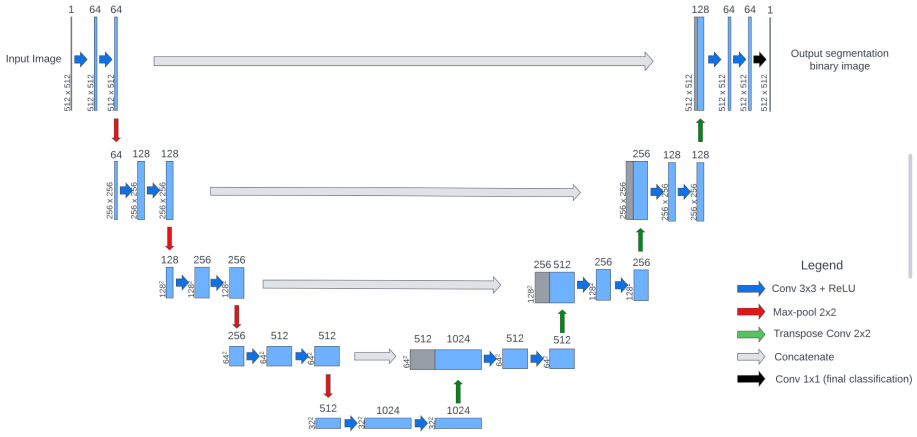

Figure 2.15: The U-Net, showing its U-shaped architecture that gave rise to the name. The grey arrows mark the concatenation links, where a down-sampling block's output is concatenated to its corresponding up-sampling block's input. This figure has been heavily inspired by the illustration in the original U-Net paper [1].

2.10 Evaluation

2.10.1 Evaluating a Deep Learning Model

Evaluation is usually done by testing the model on the test set, which consists of labeled data that the model has not seen at all during the training [14]. The output from each input in the test set is recorded and compared to the known labels. Then, by finding the numbers of true positives, true negatives, false positives and false negatives, values for some *evaluation metrics* (sometimes referred to as *scores*) can be computed [14].

2.10.2 Evaluation Metrics

These common evaluation metrics are used for many ML applications [14]. In the case of binary classification where the classes are $y \in 0, 1$, the classification results behave like this, as displayed in the confusion matrix in Figure 2.16:

- True Positive (TP): The model predicted $\hat{y} = 1$ where the ground truth was $y = 1$.
- True Negative (TN): The model predicted $\hat{y} = 0$ where the ground truth was $y = 0$.
- False Positive (FP): The model predicted $\hat{y} = 1$ where the ground truth was $y = 0$.
- False Negative (FN): The model predicted $\hat{y} = 0$ where the ground truth was $y = 1$.

For image segmentation tasks, this is commonly done on a pixel level (i.e. does a pixel belong to an object of interest or not). When these are found for all data in the test set, they can be compiled into some metrics. In the equations, the abbreviations (TP, TN, FP, FN) will refer to total numbers that were found after evaluating the models on the test set (e.g. TP will refer to the number of TPs found).

Precision

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.28}$$

Precision is in the range of $[0, 1]$ and penalizes false positives [14]. If there are no false positives, any amount of true positives will yield the ideal value of 1. If no true positives are found, the precision will be 0. The precision metric does not take any negatives into account. An overly careful classification that only classifies very few points as positives (resulting in large amounts of false negatives) may still find a very high or even perfect precision.

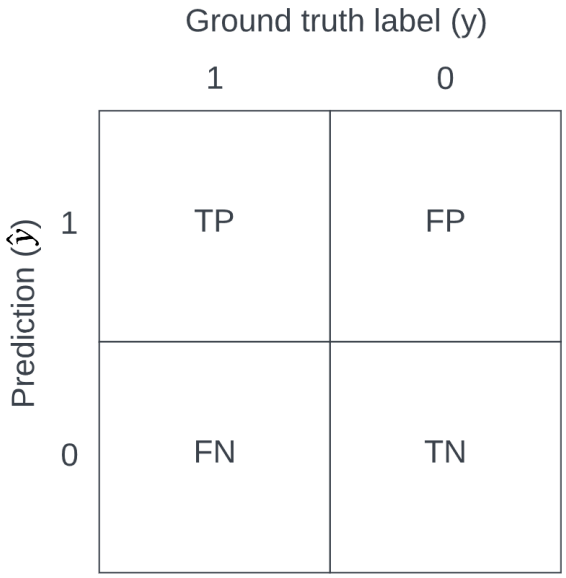

Figure 2.16: A confusion matrix for binary classification, showing the relationship between predictions and ground truth in terms of TP, TN, FP and FN.

Recall

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.29}$$

Recall is a complement to precision, is also in the range of $[0, 1]$ and penalizes false negatives [14]. A similar issue exists with recall: if you simply classify everything as positive, the recall will have the ideal value of 1. therefore, while precision and recall can be valuable metrics, they are not useful on their own.

Accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.30}$$

Accuracy is simply a metric that gives the fraction of correct classifications in the range $[0, 1]$ [14]. If 80% of the classifications were correct, the accuracy will be 0.8. This is a helpful metric since it looks at everything, but does not account for the class frequency. For example, if the "negative" class is much more common than the "positive", one can get a very high accuracy by classifying everything as negative.

F1 and Dice

$$\text{F1} = 2\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{2.31}$$

The F1-score is the harmonic mean of the precision and the recall, acting as a metric that combines the two [14]. In binary contexts, this the same as that of the *Dice Coefficient* and is a very common metric for evaluating biomedical image segmentation results [43].

Jaccard Coefficient/Intersection over Union (IoU)

$$\text{IoU} = \frac{TP}{TP + FN + FP} \tag{2.32}$$

The Jaccard coefficient, also referred to as intersection over union (IoU) is a metric that is mostly used for evaluating object detection and image segmentation solutions [43]. The metric compares the similarity of two sets A and B by computing $\frac{A \cap B}{A \cup B}$. Here, the metric is the division of the intersection between the predicted positives and the ground truth positives by the union of the predicted positives and the ground truth positives. This is illustrated in Figure 2.17. This metric can be helpful in making sure that the predicted shape of an object of interest is similar to the ground truth. IoU, like the Dice Coefficient, is commonly used in biomedical image segmentation contexts [43].

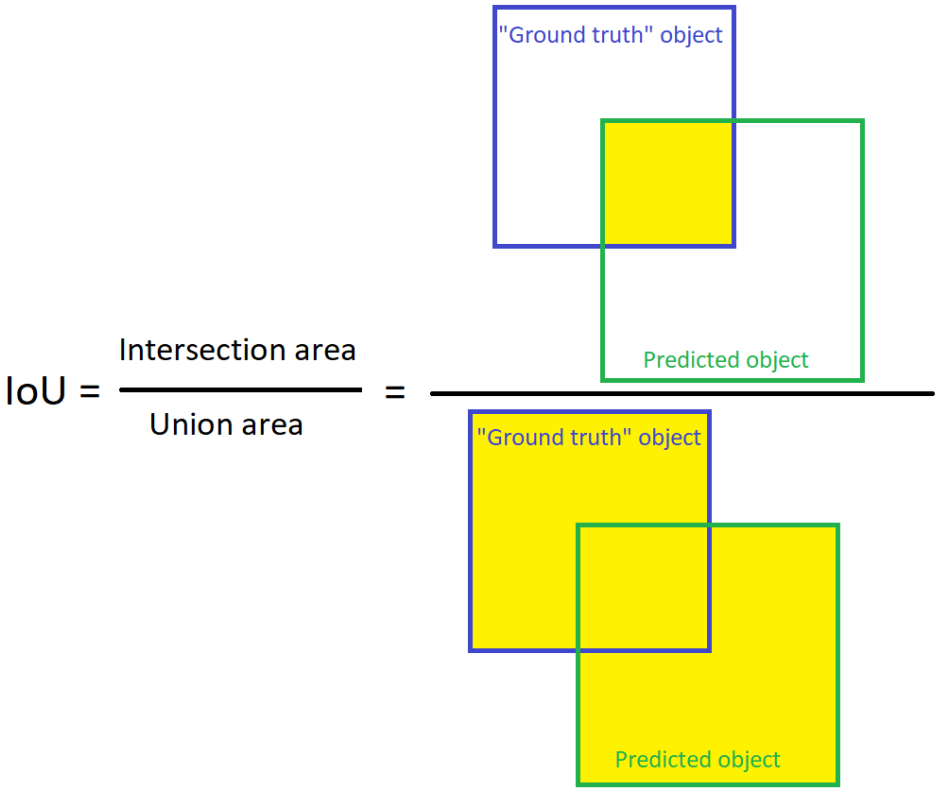

Figure 2.17: An illustration of the Jaccard coefficient (IoU) in an object detection context: the intersection (TP, the yellow area in the numerator) of the ground truth object area and the predicted object area, divided by the union (TP + FN + FP, the yellow area in the denominator) of the ground truth object area and the predicted object area.

Chapter 3

Data Background: Microscopy and Image Analysis

This Chapter will explain what kind of data was used in this project and give some insight in how said data can be processed.

3.1 Data Generation

The data used in this project derives from human cells cultured in laboratory conditions and imaged using two different microscopy techniques: phase contrast microscopy and fluorescence microscopy.

3.1.1 Cell culture

Cell lines

Cells used in this project are from so-called *cell lines*, meaning cells that grow and divide without limit as long as nutrients are available, and therefore can easily be kept in culture in the laboratory. Cell lines can be derived from a number of body cell types, such as cells from skin, connective tissue, and various organs, and are commonly used as experimental models. Cell lines can differ a lot in shape and appearance, which motivates testing our methods on multiple cell lines.

Nuclei

The nucleus is a part of the cell that stores the vast majority of the DNA [6]. Since each cell has one nucleus, segmenting the nuclei can be done instead of segmenting entire cells for the purpose of cell counting and tracking.

3.1.2 Phase-Contrast Microscopy

Phase-contrast microscopy is an old technique from the early 1930s that records phase-shifts in light that passes through some medium and translates it to brightness variations in an image. When the light travels through the cells, there is a slight phase-shift, resulting in some brightness shift in the image [8]. An example of an image generated through phase-contrast microscopy is shown in Figure 3.1.

Note that the raw image is accompanied by an enhanced version of itself that has been manually processed for displaying purposes.

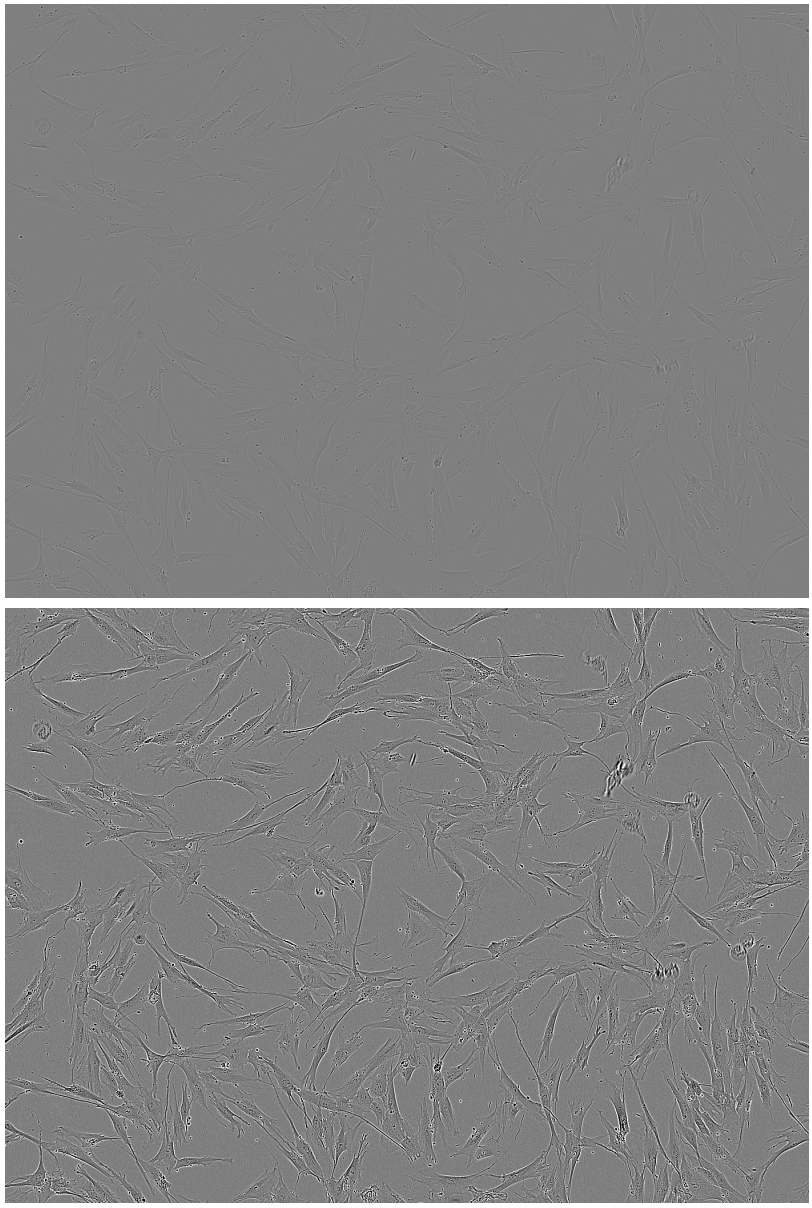

Figure 3.1: A randomly selected phase-contrast image from our data. In this image, cells from the BJ-TERT cell line are shown. (**a**): The raw image. (**b**): The same as in (a), but manually enhanced for display purposes.

Phase-contrast images can be difficult to analyze and segmenting individual cells in this kind of images has proven to be a very difficult task [44].

3.1.3 Fluorescence Microscopy

Fluorescence microscopy is a microscopy technique that utilizes *fluorescent stains* to make some cellular features more visible [6]. These are compounds that, when subject to light of some wavelength, will emit light in a different wavelength. The microscope can filter out the light of different wavelengths, focusing on the light that is emitted by the compound, and produce an image that consists of the intensities of the emitted light alone. Some of these compounds are molecules that bind to DNA. Since DNA is for the most part concentrated to the nuclei of cells, these compounds can be used to display nuclei in microscopy images much more clearly than what phase-contrast microscopy does. These compounds are often referred to as *nuclear dyes* or *nuclear stains* because of this [6].

To show the power of this, Figure 3.2 displays the same cells as Figure 3.1 (images taken at the same place, at the same time-point) when using fluorescence microscopy instead of phase contrast.

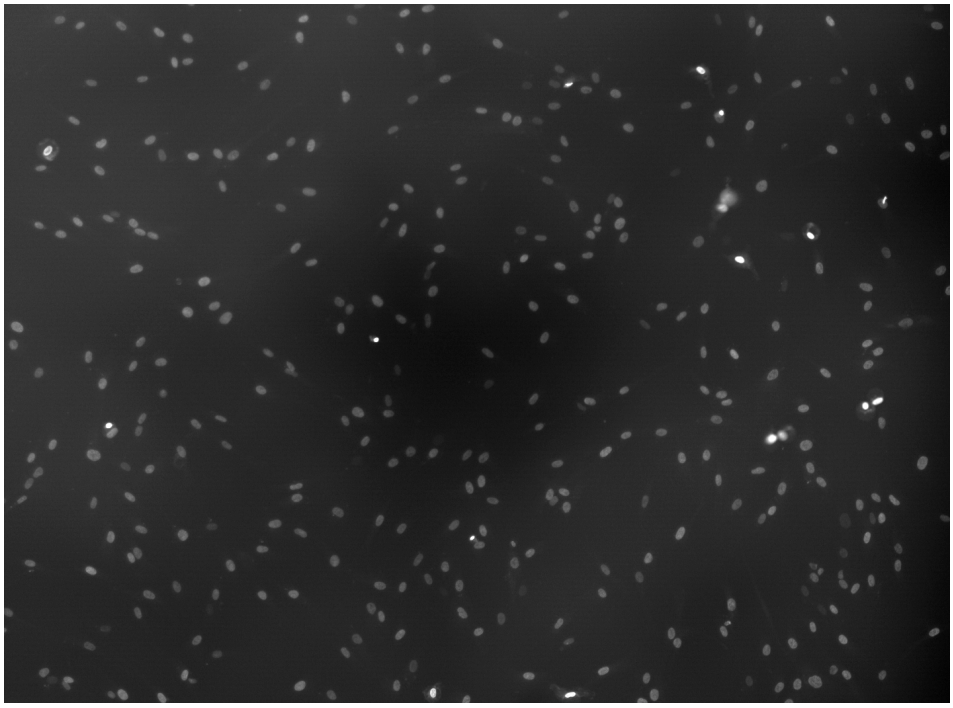

Figure 3.2: An image generated using fluorescence microscopy. The image has been manually enhanced for visibility. The small, bright "blobs" mark the nuclei of the cells.

While fluorescence microscopy solves a lot of problems for processing and segmentation purposes, fluorescence microscopy comes with a huge flaw. The high

intensity light exposure and the nuclear stains themselves are toxic to the cells, causing cells to behave differently and possibly die [7]. Because of this, fluorescence microscopy is not viable for live-cell imaging when performing experiments on cells for long time lapses.

3.2 Data Processing

Before analyzing images, it is common to do some pre-processing to make the image data easier to analyze. This can be done for several purposes, such as removing noise, enhancing contrasts or transforming the image data to some desired format.

Images from fluorescence microscopy can be very dark, and might therefore need some sort of processing before any meaningful information can be extracted from them.

3.2.1 Image Histograms

An image histogram plots the distribution of light intensities in an image. In other words, it shows how many pixels belong to each possible light intensity, limited by the bit-depth of the image. In microscopy images, these histograms are often very concentrated around a certain point, as shown in Figure 3.3. Contrast can be enhanced by normalizing the pixel values. An issue with histogram equalization is that if the image has some areas that are significantly darker or lighter than most other areas, they may not be enhanced in a desirable way. To deal with this, local contrast enhance algorithms have been proposed.

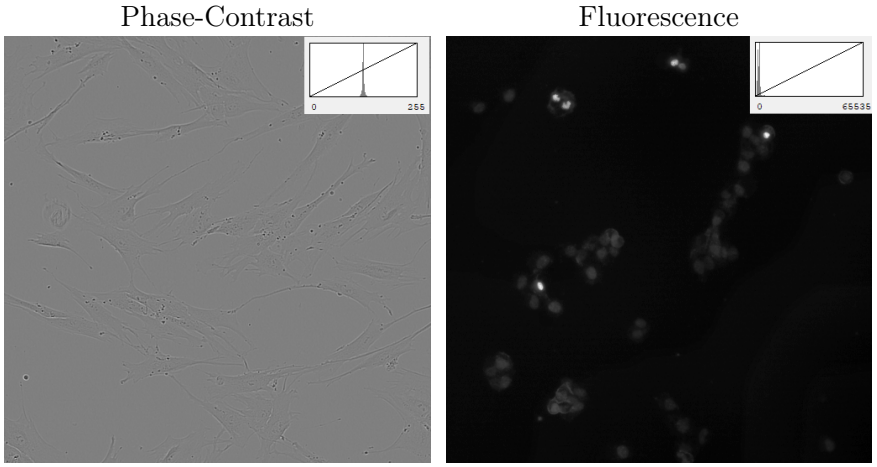

Figure 3.3: Phase-contrast (left) and fluorescence (right) microscopy images along with their corresponding image histograms.

3.2.2 Adaptive Histogram Equalization (AHE)

AHE is an algorithm that attempts to enhance contrasts in the image by looking at different sections of the image and computing histograms for each section. The histograms are then equalized independently, enhancing contrasts in areas of the image independent of each other [45]. This limits the effect that high-intensity outliers have on the overall histogram equalization but may also amplify noise significantly in parts where the regional histogram is very concentrated (i.e. homogeneous parts where most pixels have the same value).

3.2.3 Contrast-Limited Adaptive Histogram Equalization (CLAHE)

Like the name suggests, CLAHE is a version of AHE that limits the contrast amplification [46]. This is done to counter the noise enhancement that AHE sometimes provides. CLAHE has been proven to produce good results on medical images and has therefore been used for contrant enhancement in bioinformatics [46].

3.2.4 Thresholding

Thresholding is a very simple segmentation technique that makes an image binary [47]. This is typically done by setting some threshold T, where every pixel whose light intensity is above T gets assigned the value 1 (or 255 in the case of an 8-bit image) and every pixel whose light intensity is below T get assigned the value 0, thus making the image consist solely of two different values. This can be useful when segmenting an image, but global thresholding techniques may yield insufficient results due to intensity variations in the image. Setting T too high will include noise in the segmentation, setting T too low might exclude noise but also some of your objects of interest [47].

Local thresholding, similar to adaptive histogram equalization, attempts to solve issues that arise when processing entire images at once by splitting it into smaller regions and processing them independently.

3.2.5 Otsu's Method

This is an overview of Otsu's method, for more details, see the original paper [47]. Otsu's method is a thresholding algorithm that can be used on a global or local scale. It looks for an optimal threshold T by minimizing the intra-class variance $\sigma_w^2(T)$. This variance is defined as a weighted sum of the intra-class variances in each class. For a given threshold T, if $N_0(T)$ is the amount of pixels in the 0-class, $N_1(T)$ is the amount of pixels in the 1-class and N_{tot} is the total amount of pixels, the weights ω_0 and ω_1 are defined as [47]:

$$\begin{aligned} \omega_0(T) &= \frac{N_0(T)}{N_{tot}} \\ \omega_1(T) &= \frac{N_1(T)}{N_{tot}} \end{aligned} \tag{3.1}$$

Then, the intra-class variance $\sigma_w^2(T)$ is defined as [47]:

$$\sigma_w^2(T) = \omega_0(T)\sigma_0^2(T) + \omega_1(T)\sigma_1^2(T) \tag{3.2}$$

Where σ_0^2 and σ_1^2 are the intra-class variances for the 0-class and the 1-class respectively.

The algorithm tests all different thresholds $T = 1, 2, ..., T_{max}$. Since the amount of pixels belonging to each class and the mean intensity of each class changes with each value for T, the weights and means will need to be computed for each value for T [47]:

$$\begin{aligned} \mu_0(T) &= \frac{\sum_{k=0}^{T-1} kp(k)}{\omega_0(T)} \\ \mu_1(T) &= \frac{\sum_{k=T}^{L-1} kp(k)}{\omega_1(T)} \end{aligned} \tag{3.3}$$

Where $p(k)$ is the amount of pixels with intensity k divided by the total amount of pixels N_{tot}. Using these values for each T, inter-class variance $\sigma_b^2(T)$ can be computed for each T [47]:

$$\sigma_b^2(T) = \omega_0(T)\omega_1(T)(\mu_0 - \mu_1)^2 \tag{3.4}$$

Maximizing the inter-class variance $\sigma_b^2(T)$ is equivalent to minimizing $\sigma_w^2(T)$ [47]. Therefore, the algorithm computes $\sigma_w^2(T)$ for all values of T and finds the T that yields the highest value for $\sigma_b^2(T)$. This value for T is then used for the thresholding.

A cell image, processed using Otsu thresolding, is displayed in Figure 3.4.

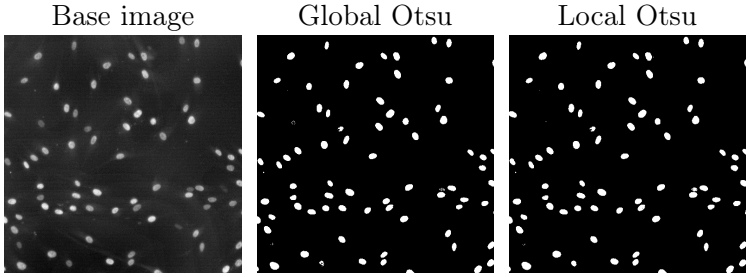

Figure 3.4: A cell image from fluorescence microscopy, processed using global and local Otsu segmentation.

3.3 Data Processing for Machine Learning

In this section, some concepts are brought up that are explained in Chapter 2.

3.3.1 Binary Image Masks

A thresholded, binary image can be used as an *image mask*. A binary image mask refers to something that defines a region of interest in an image. Images like these

can be very useful when training a machine learning model for image segmentation tasks (more in Chapter 2), as they can serve as labels for raw image data.

3.3.2 Data Augmentation and Synthetic Data

Data augmentation is the process of creating *synthetic data* from real data. For images, this can be achieved by modifying the images in some way. In this project we have resorted to using simple augmentation techniques, such as the following transformations:

Height and Width Shifts

Shifts can be applied to an image matrix A of dimensions m x n by applying a transformation through a shift matrix S. Shift matrices are modified diagonal matrices, consisting of ones on some diagonal (not the main diagonal) and zeroes elsewhere, such as the subdiagonal matrix S_l of dimension m x n:

$$S_l = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix} \tag{3.5}$$

A vertically shifted matrix A_{vs} can be obtained by using the transformation $S_l A$:

$$\begin{aligned} S_l A &= \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,2} & \dots & a_{m-1,n-1} & a_{m-1,n} \\ a_{m,0} & a_{m,2} & \dots & a_{m,n-1} & a_{m,n} \end{bmatrix} \\ A_{vs} = S_l A &= \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,2} & \dots & a_{m-1,n-1} & a_{m-1,n} \end{bmatrix} \end{aligned} \tag{3.6}$$

Likewise, a horizontally shifted matrix A_{hs} can be obtained by using the transform AS_l:

$$AS_l = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,2} & \dots & a_{m-1,n-1} & a_{m-1,n} \\ a_{m,0} & a_{m,2} & \dots & a_{m,n-1} & a_{m,n} \end{bmatrix}$$

$$A_{hs} = AS_l = \begin{bmatrix} a_{1,2} & \dots & a_{1,n-1} & a_{1,n} & 0 \\ a_{2,2} & \dots & a_{2,n-1} & a_{2,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,2} & \dots & a_{m-1,n-1} & a_{m-1,n} & 0 \\ a_{m,2} & \dots & a_{m,n-1} & a_{m,n} & 0 \end{bmatrix} \tag{3.7}$$

The extent of the shifts can be adjusted by changing which diagonal is filled with ones in S. By transforming with the transpose $S^\top$ instead, the shift will be done in the opposite direction. The shift transformation leaves the resulting matrix with blank space (zeroes). This can be handled in different ways and one such way will be described in this section. Vertical and horizontal shifts are displayed in Figure 3.5.

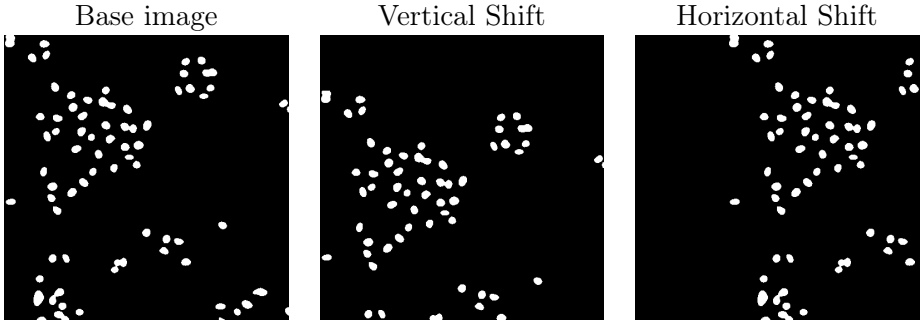

Figure 3.5: A binary image mask, accompanied by vertically and horizontally shifted versions of itself.

Vertical and Horizontal Flips

Flipping an image matrix A is done by reversing the order of rows (vertical flip) or columns (horizontal flip). This can be achieved by using a linear transformation, multiplying A with a row-reversed identity matrix I_r:

$$I_r = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & \dots & 1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 1 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \end{bmatrix} \tag{3.8}$$

The vertically flipped image matrix A_V is found using the transformation $I_r A$:

$$I_r A = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & \dots & 1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 1 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,2} & \dots & a_{m-1,n-1} & a_{m-1,n} \\ a_{m,0} & a_{m,2} & \dots & a_{m,n-1} & a_{m,n} \end{bmatrix}$$

$$A_V = I_r A = \begin{bmatrix} a_{m,0} & a_{m,2} & \dots & a_{m,n-1} & a_{m,n} \\ a_{m-1,0} & a_{m-1,2} & \dots & a_{m-1,n-1} & a_{m-1,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \end{bmatrix} \tag{3.9}$$

Similarly, the horizontally flipped image matrix A_H is found using the transformation AI_r:

$$AI_r = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,2} & \dots & a_{m-1,n-1} & a_{m-1,n} \\ a_{m,0} & a_{m,2} & \dots & a_{m,n-1} & a_{m,n} \end{bmatrix} \begin{bmatrix} 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & \dots & 1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 1 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \end{bmatrix}$$

$$A_H = AI_r = \begin{bmatrix} a_{1,n} & a_{1,n-1} & \dots & a_{1,2} & a_{1,1} \\ a_{2,n} & a_{2,n-1} & \dots & a_{2,2} & a_{2,1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,n} & a_{m-1,n-1} & \dots & a_{m-1,2} & a_{m-1,1} \\ a_{m,n} & a_{m,n-1} & \dots & a_{m,2} & a_{m,1} \end{bmatrix} \tag{3.10}$$

A binary image, along with its vertically and horizontally flipped versions, are displayed in Figure 3.6.

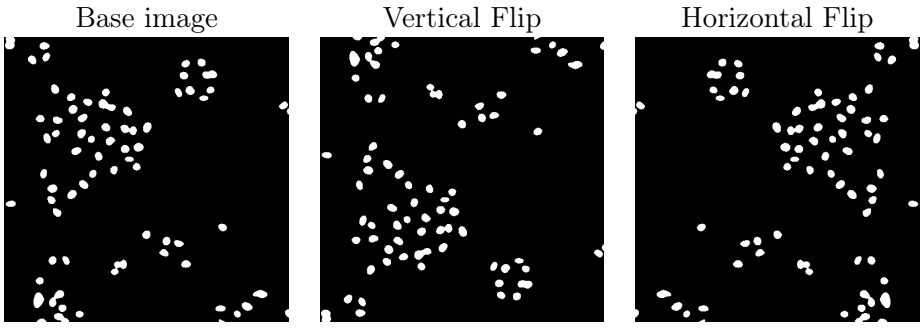

Figure 3.6: A binary image mask, accompanied by vertically and horizontally flipped versions of itself.

Rotation

Rotation can be done by multiplying each point p in the image matrix with a rotation matrix R. If the center of the image is viewed as the origin, the image can be rotated θ degrees counterclockwise in-place using this transform:

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = Rp = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \tag{3.11}$$

This makes (a', b') the new coordinates for the pixel at (a, b). Just like shifts, this operation can leave blank spaces in the resulting image matrix. A rotation example is shown in Figure 3.7.

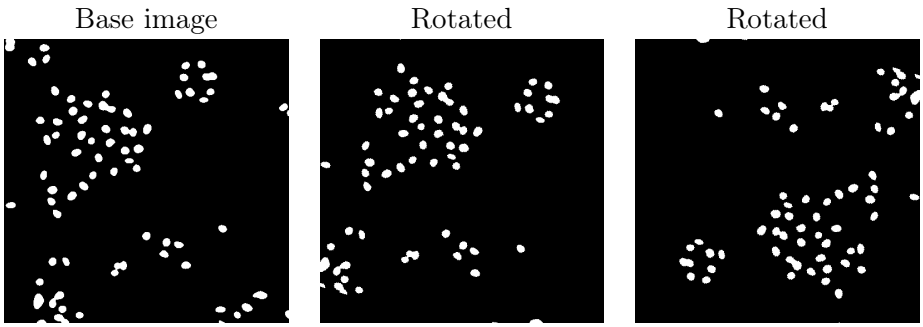

Figure 3.7: A binary image mask, accompanied by rotated versions of itself.

Shearing

Shearing can be done, like rotation, by multiplying each point p in an image matrix with a matrix. Shear matrices are commonly in the form of $S = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$ for horizontal shears or $S = \begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$ for vertical shears. So, the new coordinates (a', b') for a pixel at coordinates (a, b) after a shear becomes (for a horizontal shear):

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = Sp = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \tag{3.12}$$

This also leaves blank spaces. Examples of vertical and horizontal shears are displayed in Figure 3.8.

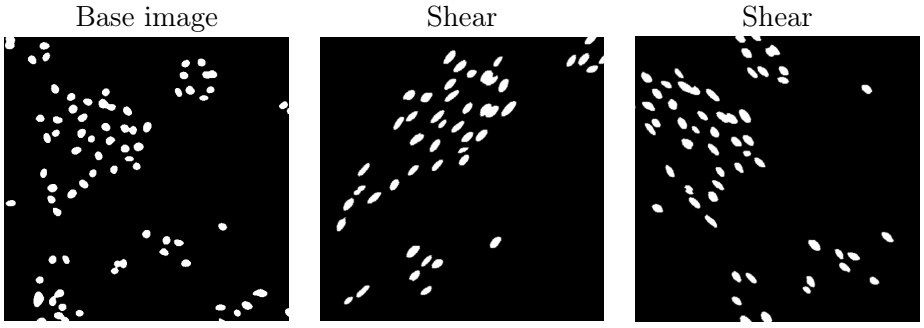

Figure 3.8: A binary image mask, accompanied by two randomly sheared images.

Dealing with Blank Spaces: Reflection

One way to deal with the blank spaces that can be left behind by augmentations is by using reflection, i.e. letting the blank space be filled with a reflection of the part of the image that leads to the edge. This can be useful when having images that contain multiple small objects of interest since they will rarely get distorted. When augmenting images such as photographs, reflection might not be desired since objects near the edges will appear in ways they will never appear in real data. Reflection is shown in Figure 3.9.

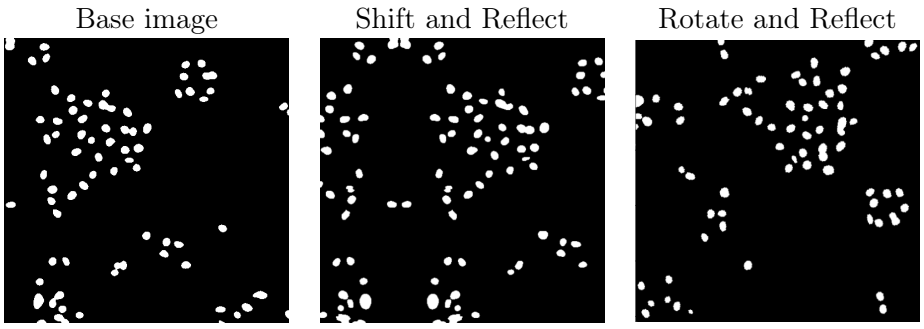

Figure 3.9: A binary image mask, accompanied by a shifted and a rotated image where the blank spaces have been filled using reflection.

3.3.3 Applying Data Augmentation in Training

With data augmentation, one can expand their dataset which may improve training results. This may also serve as a sort of regularization [48]. Data augmentation can be applied to a dataset before training or during training. When used before training, a fixed amount of randomly generated augmentations are usually applied to the training set, expanding it. When used during training, a random mix of augmentations can be applied every time an entry is being processed. Examples

of augmented images (with a random combination of augmentations) are shown in Figure 3.10.

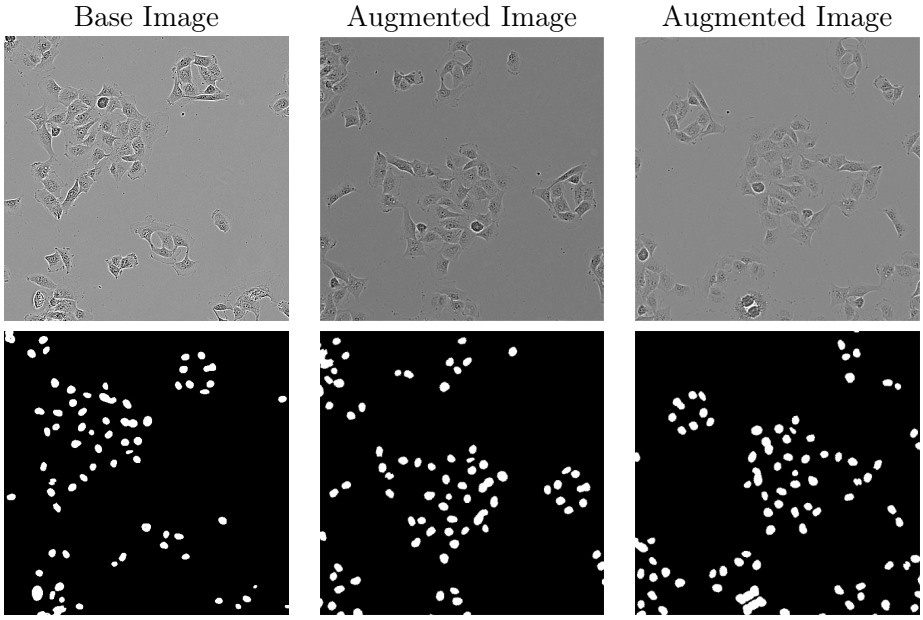

Figure 3.10: Augmented training data samples. The leftmost column shows a phase-contrast image with its corresponding binary image label without any augmentation applied. The following columns show two image pairs where two different, randomly selected combinations of augmentations (here: shift, shear, flips, rotation and filling leftover space with reflections) have been applied to the base images. Note that the both images in each pair has the same combination of augmentations to them. The phase-contrast images have been manually enhanced for visibility.

Chapter 4

Software and External Resources

In this chapter, I will explain background and details about the technical implementation tools that were used in the project.

4.1 Python

Most of the code that was written for this project, including all deep learning algorithms, was done so in Python (version 3.10.6). Python has become the most widely used programming language for ML and Data Science [49] due to its powerful yet accessible nature. Because of its popularity, several frameworks have been developed to make programming complex ML models a much more accessible task.

4.1.1 Tensorflow and Keras

For the deep learning implementation, I used Tensorflow (version 2.10.0) and Keras (version 2.10.0). Tensorflow is an open source ML library developed by Google [50] and Keras is an open source framework that was built upon Tensorflow for deep learning implementations [51]. To utilize Nvidia GPUs for training, I used CUDA (version 11.2), Nvidia's toolkit for building GPU-accelerated applications [52].

4.1.2 Anaconda

For package management, I used Anaconda (version 22.9.0), the most widely used Python distribution [53]. Packages were installed using conda-forge, a community-driven collection of repositories for Anaconda [54].

4.2 Fiji and ImageJ

Some image processing was done using Fiji (version 2.9.0): a distribution of the open-source image processing software ImageJ, focused on biological image analysis [55]. ImageJ is developed at the National Institution of Health (NIH). The software has been a foundation for image analysis in biological imaging for many years and continues to develop through community collaborations. It is scriptable and can be used for programming automatic image processing pipelines [56].

4.3 CellProfiler

Other image processing was done using CellProfiler [57]. CellProfiler is an open-source cell image analysis software that was developed at the Broad Institute of MIT and Harvard. It supports building custom pipelines for processing cell images and extracting data.

4.4 C3SE Alvis (NAISS)

The computation cluster Alvis was used to train the deep learning models in this project. The hardware (primarily GPUs) used to train the models in this project was all part of the Alvis cluster and accessed remotely. Alvis is dedicated to AI and ML research and is a part of Chalmers Center for Computational Science and Engineering (C3SE), a centre for scientific and technical computing at Chalmers University of Technology. Alvis is also a part of the National Academic Infrastructure for Supercomputing in Sweden (NAISS). [58].

Chapter 5
Methodology

This chapter will describe the methods that we used, from generating data to applying trained deep learning models to identify nuclei. The general method as well as experimental specifics will be included here. Specific cell lines will be introduced and a deep understanding of what they are might be outside the scope of this thesis. As a reader, it is only important to understand that cells from different cell lines are morphologically different and may have some unique traits. The code that we wrote for training, evaluating and using our deep learning models can be accessed at `https://github.com/Nilsson-Lab-KI/cell_identification`.

5.1 The Pipeline

To achieve automatic cell identification and apply it for high content experimental data, the general pipeline was this:

1. Obtain raw data from phase-contrast and fluorescence microscopy (Section 3.1.1).
2. Process the fluorescence images and convert them into binary images.
3. Build datasets, using raw phase contrast images as X and the binary images as the labels Y.
4. Train a U-Net (Section 2.9) for nucleus identification using these datasets.
5. Use the trained U-Net to predict nucleus locations in phase-contrast time-lapses.
6. Application: count the nuclei in the predictions for cell counting, apply cell tracking algorithms for cell tracking.

An overview of this is shown in Figure 5.1. The following sections will go into detail for how the first five steps were done in this project. Since the last step is more of an application of our project it will not be discussed here. However, since it was very interesting for us, we explored possibilities for cell tracking. What we found regarding cell-tracking can be found in the future work Section 8.1.

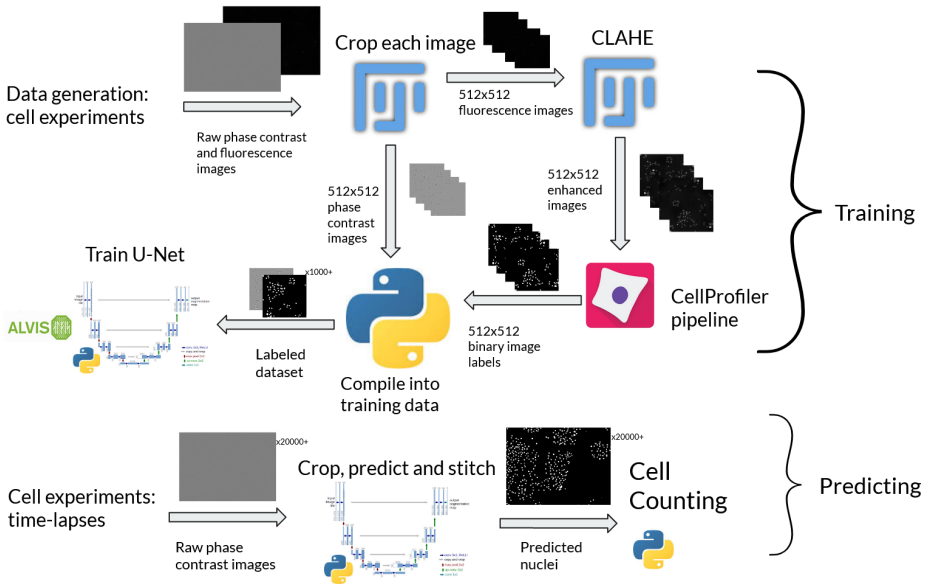

Figure 5.1: Method overview, from acquiring the data to extracting data from U-Net-predicted nuclei with a phase-contrast time-lapse as input. Python, CellProfiler, Fiji and Alvis logos are taken from their respective websites.

5.2 Making Training Datasets

The datasets that were used in this project were made in-house using live-cell imaging microscopy techniques. All datasets consisted of entries of *labeled* phase-contrast microscopy images, which were made of phase-contrast images (X) that were accompanied labels in the shape of binary image masks (Y) that marked the positions of the nuclei. An example is shown in Figure 5.2. This section will explain how these were made.

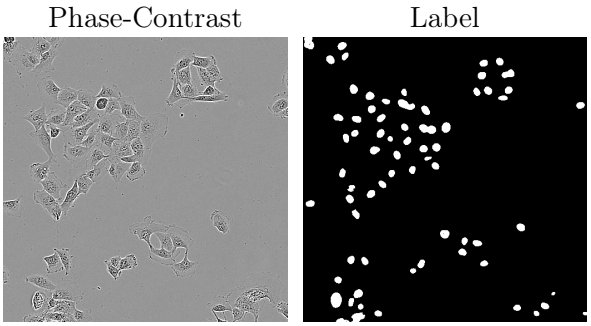

Figure 5.2: A phase-contrast image (left) and its assigned label (right), the kind of labeled data that our datasets consisted of.

5.2.1 In-House Microscopy

Experiments were performed by monitoring live, cultured cells. For the sake of generating training data, a nuclear dye was used, enabling the use of fluorescence microscopy. At certain time points, images were saved using both phase-contrast (Section 3.1.2) and fluorescence (Section 3.1.3), giving two versions of each cell image. Since a nuclear dye was used in the fluorescence microscopy, these images mainly showed nuclei. Specific information regarding these cell experiments is explained in Section 5.6.

In the cell experiments, cultured cells were left to grow in separate containers called *wells*. Each well was part of a n x n grid that split the wells into n^2 *tiles* as shown in Figure 5.3. Cells could not exit their wells but could freely move between tiles. The microscope imaged each tile separately, resulting in one image per tile, per well, per time-point.

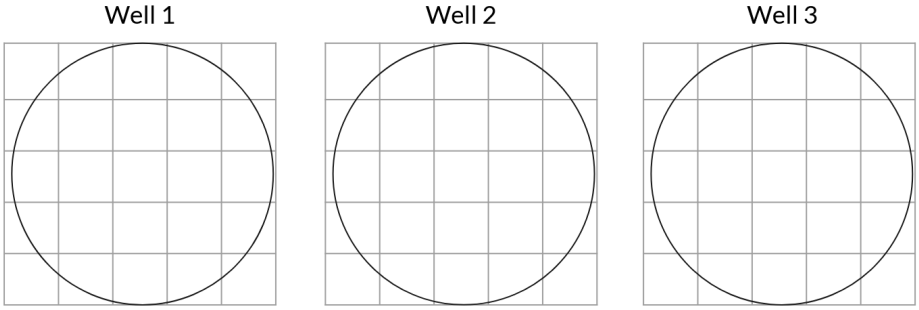

Figure 5.3: A rough illustration of a cell experiment structure for microscopy, consisting of cells in 3 wells where each well is split into 25 separate tiles.

5.2.2 Data Specifications

The images from phase-contrast microscopy were saved as 1408x1040-sized 8-bit grey-scale .tiff files. The images from fluorescence microscopy were saved as 1408x1040-sized 32-bit .tiff files. These images were cropped into 4 different 512x512 images. To ensure that we still had image-pairs that were suitable as training data, the crops were always taken at the same locations in the phase-contrast images as the fluorescence images.

5.2.3 Data Processing Pipeline

The fluorescence images were processed in two steps: preparation and segmentation. The steps of this process are illustrated in Figure 5.4.

Fluorescence Image Preparation

The preparation step was made by applying CLAHE (Section 3.2.3) to the cropped images using a script that we made for the ImageJ software. CLAHE was ap-

plied using a block size of 127, 256 histogram bins and a maximum slope of 3.0. This simple approach was chosen after trying various, more complex pipelines and finding that they did not adapt well to the light-intensity variation found in the fluorescence images. Since we wanted a script that could process all of our data, an approach that yielded great results on some of the data but bad results on the rest could not be used. Thus we opted to just using CLAHE.

Fluorescence Image Segmentation

In the segmentation step, we wanted to convert the prepared fluorescence images into binary images that we could use as labels for the training data. Starting off, we attempted to build a macro in ImageJ to do this using a mix of filters and thresholding techniques. After getting unsatisfactory results, we instead built a custom pipeline in the CellProfiler software which thresholded the images and captured the nuclei using the `IdentifyPrimaryObjects` module. We customized this module with the following parameters (if a parameter is not listed, default values were used):

- Typical diameter (pixels): 5-30
- Discard objects outside the diameter range: Yes
- Thresholding strategy/method: Adaptive (Local) Otsu (Section 3.2.5)
  - Bounds on threshold: 0.05-1.0
  - Size of adaptive window (pixels): 20x20

Just like in the preparation step, we wanted to create a robust, flexible pipeline that could yield good results for all our data, rather than perfect on some and terrible on the rest.

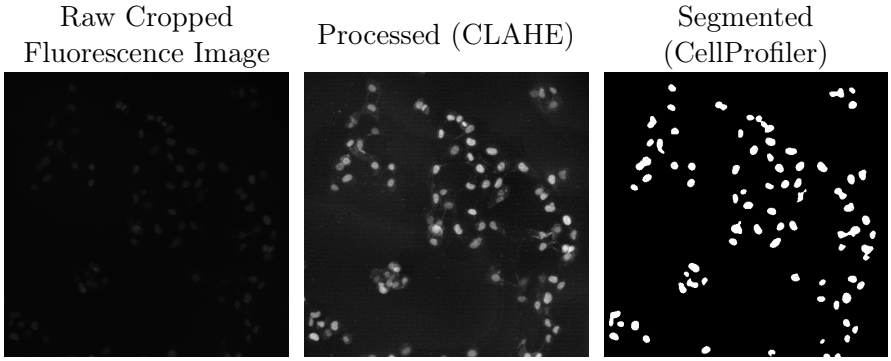

Figure 5.4: Illustration of the steps that were taken (from left to right) for converting raw fluorescence images into binary image masks that could be used as labels. First, a crop from a raw fluorescence image which is very dark. Then, after applying CLAHE, many nuclei are clearly visible, but some noise is amplified too. Finally, after segmenting the processed image with CellProfiler, the image is converted into a binary image.

5.2.4 From Processed Data to Training Data

Some fluorescence image had strange properties (stemming from e.g. noise, the microscope being out of focus, artifacts from humidity or nuclei that did not respond to the dye) which could lead to extreme cases where the fluorescence image processing pipeline could have unexpected output. Some of the resulting labels could consist of a large amount of segmented noise and some could be completely devoid of segmented nuclei. To deal with this, we applied an automatic filtering approach to the processed data. This filtering process removed a fixed part of the dataset that consisted of labels that had a low amount of nuclei as well as labels that had an unreasonably high amount of nuclei. The cutoff points were computed for each dataset, but a reasonable starting point (initially found by manually reviewing the datasets) seemed to be to remove the images in the lowest 25% in terms of nucleus-pixels and any images that had a number of nucleus-pixels that was higher than the number of background pixels, since this was almost always an effect of segmented noise. This was done to avoid including images that had not been processed well by our data processing pipeline, as well as images of cells that had not responded well to the nuclear dye, since such images could have a negative impact on the training. Some examples of labels in entries that would be filtered out automatically thanks to this are shown in Figure 5.5.

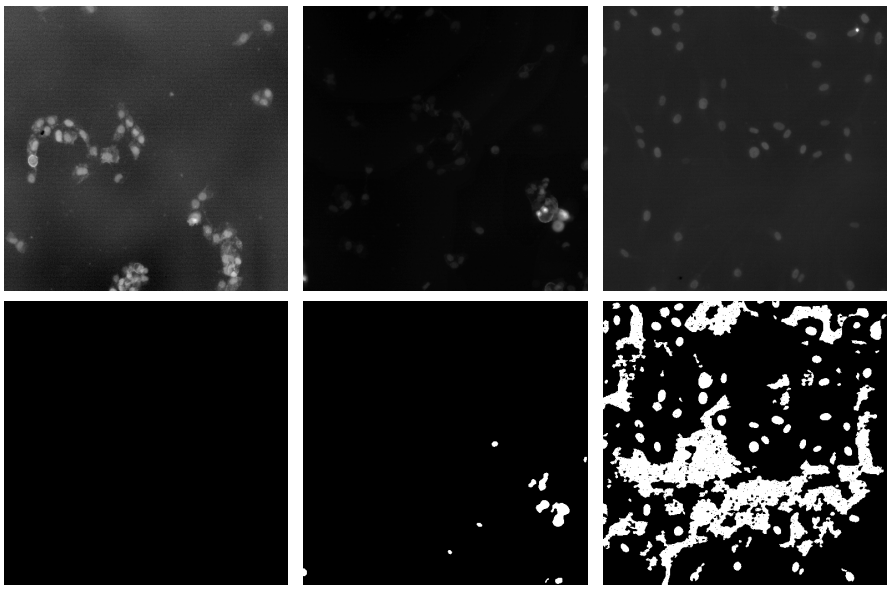

Figure 5.5: Extreme cases of unexpected behaviour by our image processing pipeline and the reason why we applied automatic filtering. **(a)**: Processed fluorescence images. **(b)**: Poorly segmented labels, generated by applying our CellProfiler pipeline to the images in (a).

To see the effects of this automatic filtering approach, we ran experiments

with automatically filtered data as well as manually filtered data. The manually filtered data was made by manually comparing each binary label in a dataset to its fluorescence image source and discarding the image pair if the binary label had any visible errors. When comparing the results of the two approaches, we did not find any significant differences. Thus we opted to moving forward with automatically filtered data.

5.3 Building the Deep Learning Models

We opted to using the U-Net (Section 2.9) as our deep learning model because of its proven success in biomedical image segmentation tasks [42].

5.3.1 U-Net Implementation Specifics

For information regarding hyperparameters, see Section 2.3 and Section 2.7.6. Most of the structural hyperparameters (kernel size, amount of kernels, activation functions, weight initialization, padding, stride) were taken from the original U-Net paper [1]. A low dropout of 0.05 was chosen across the network since the U-Net exclusively consists of convolutional layers and may benefit from a low dropout [34]. Binary focal cross-entropy (Section 2.6.2) was chosen as loss function due to its proven performance in object detection [37]. Adam (Section 2.5.5) was chosen as optimization algorithm due to its flexibility and robustness. A low batch size was used because of technical limitations. Batch normalization (Section 2.2.10) was applied for all convolutional layers

We tested different values for α but opted to using $\alpha = \frac{1}{2}$ (no explicit class weighting) since the γ hyperparameter already applies class balancing in the sense that it lowers the impact of easy background classifications on the loss function [37]. A sample of this will be shown in the discussion Section 7.2.2.

Because we had access to a computational cluster, we used a very high epoch ceiling (far higher than any of the models would need) and a very high patience, reducing the risk of having the training terminate earlier than needed.

Because of time limitations and since original parameters worked well, we did not explore hyperparameter setups to the fullest. The final hyperparameter setup we used was:

- Convolutional layer specifics
  - Kernel Size: 3x3 (for all hidden convolutional layers, the output convolutional layer used 1x1)
  - Amount of kernels: 64 (starts at 64, doubles after each downsampling step and halves again in each upsampling step)
  - Activation function (hidden layers): ReLU
  - Padding: 1
  - Stride: 1
  - Dropout: 0.05 (for all hidden convolutional layers)
  - Weight initialization: He Normal

- Pooling: 2x2 Max Pooling
- Activation function (output layer): Logistic (sigmoid)
- Loss function: Binary Focal Cross Entropy
  - $\gamma = 2$
  - $\alpha = \frac{1}{2}$
- Optimizer: Adam
  - Learning rate: $2 \cdot 10^{-4}$
  - Rest default values.
- Batch size: 4
- Maximum amount of epochs: 3000
- Patience: 250 (reverting back to the best model)

5.3.2 Training Process

During the training, each data sample was augmented (Section 3.3.2) using a random combination of rotation, width shifts, height shifts, shear, horizontal flips and vertical flips. Blank space that was left behind by the augmentation was filled with reflection of nearby pixels.

All U-Net models were trained on the Alvis cluster using a single Nvidia A100 GPU. Most models found their minimum validation loss within 400 epochs. With the 250 epochs that were added by the patience, the training usually took between 5-15 hours depending on the dataset. Once trained, predictions took about 30ms each on the same hardware.

5.4 Evaluating the U-Net Models

Evaluating the models became challenging since our automatic labeling pipeline did not guarantee a ground truth. Simply comparing the U-Net's output to the labels was not sufficient to evaluate how well our models were doing. Furthermore, since our main interest was identifying nuclei and we were less interested in their exact shapes, a pixel-level evaluation would not be ideal. We therefore combined three different evaluation approaches: standard pixel-level comparisons with the labels, object-level comparisons with the labels to see which nuclei were captured and a manual evaluation by experts where they compared the predicted nuclei with the corresponding input phase-contrast images.

5.4.1 Evaluating on a Test Set: Pixel-Level

Here, we processed the test sets and compared the outputs to the labels, counting each predicted pixel as a true positive, true negative, false positive or a false negative. We then computed the average precision, recall, accuracy, Dice-score and IoU.

5.4.2 Evaluating on a Test Set: Object-Level

To compare the image masks y and $\hat{y}$ on an object level, we implemented an algorithm that followed these steps, shown in Figure 5.6. Here, $\odot$ refers to the Hadamard product (element-wise product).

1. Compute the overlap $TP = y \odot \hat{y}$ to find the true positive pixels.

2. Find the connected components in y and $\hat{y}$ and give each component a unique number marker (i.e one component in each image consists of 1s, one consists of 2s etc). Call these marked images y_l and $\hat{y}_l$

3. Compute the matrix multiplications $O_1 = TP \odot y_l$ and $O_2 = TP \odot \hat{y}_l$. This will give two nearly identical marked images of true positive components, where the marking convention corresponds to those in y_l and $\hat{y}_l$ respectively.

4. Loop through the marks (the assigned numbers) in y_l and $\hat{y}_l$. Count the number of marks l_i that satisfy $l_i \in y_l \wedge l_i \notin O_1$ and $l_i \in \hat{y}_l \wedge l_i \notin O_2$. Those will be the number of false negative and false positive *objects* respectively.

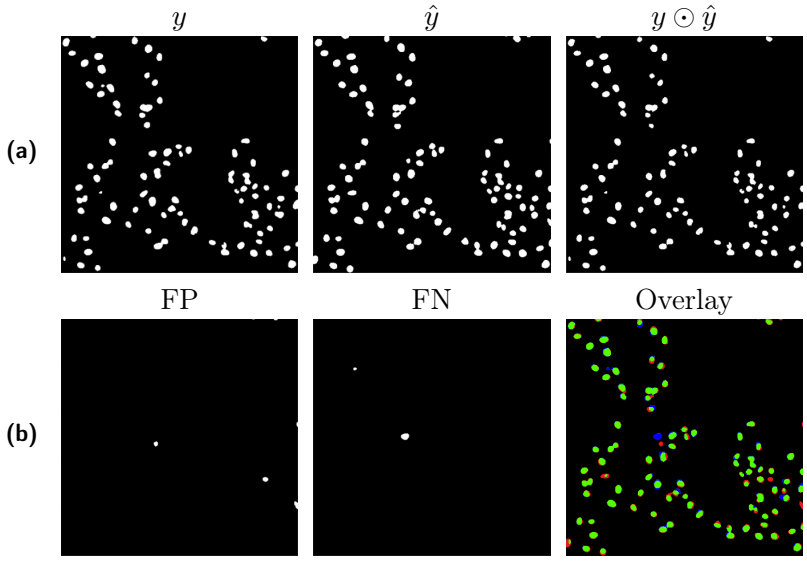

Figure 5.6: **(a)**: From left to right: a binary label, a prediction and their overlap $y \odot \hat{y}$ (pixel-level true positives). **(b)** The caught object-level false positives and false negatives, using the algorithm stated above. The rightmost image is a color-coded overlay of y and $\hat{y}$, where green represents true positive, black represents true negative, red represents false positive and blue represents false negative pixels, which can be used to verify the left and center images. This overlay image can be used to verify the FP and FN images.

Note that because this algorithm compares objects, it does not compute any amount of true negatives. Because of this, accuracy cannot be computed. These values can however give an object-level approximation of precision, recall, Dice-score and IoU.

5.4.3 Manual Expert Evaluation with Mask Overlays

Manual evaluation was done by overlaying predictions upon the corresponding phase-contrast input images, as displayed in Figure 5.7. By comparing this image with the original phase-contrast image, experts could identify false positives and false negatives by manually analysing the image. To help with this, the original fluorescence image was also included as a reference. Because this was a very tedious, time-consuming process, we randomly selected small subsets of the test sets consisting of 30 images for this. Figure 5.8 shows a phase-contrast image, a phase-contrast image with a prediction overlaid upon it and a fluorescence image together, which was the kind of data that the experts used to perform their evaluation.

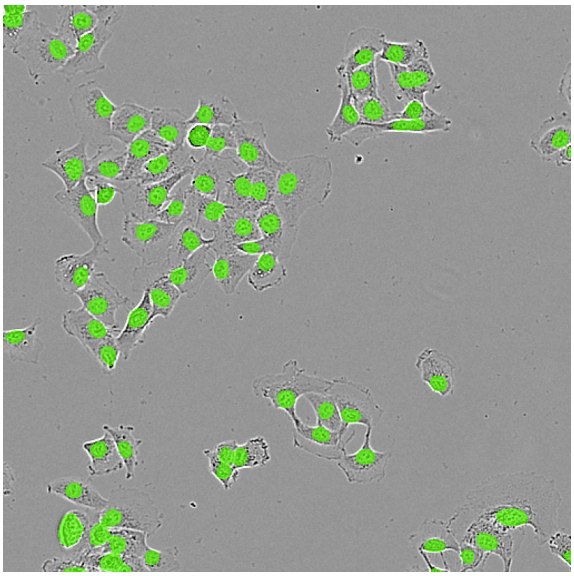

Figure 5.7: A prediction overlaid upon a phase-contrast image, like those used for manual evaluation. Green pixels represent predicted nuclei. In other words, green pixels are pixels that are white in the predicted binary segmentation image. For display purposes, the underlying phase-contrast image has been enhanced manually.

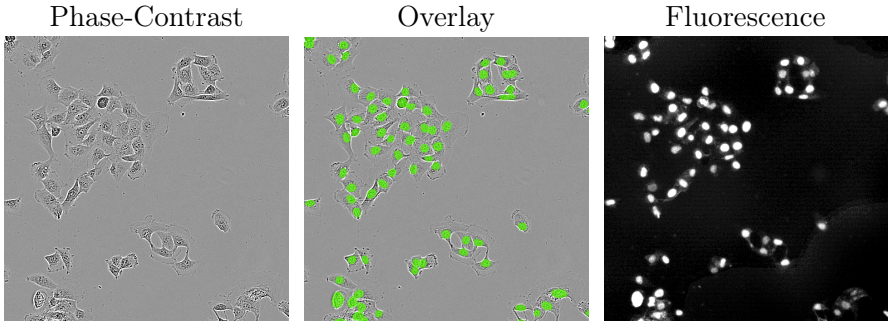

Figure 5.8: From left to right: a phase-contrast image, a phase-contrast image with predictions overlaid upon it and the corresponding fluorescence image. As usual, the microscopy images have been manually enhanced for display purposes.

5.5 Experiments

5.5.1 U-Net Models

Because of time limitations, we decided to focus on three datasets, each consisting of microscopy images of cultured cells from a specific cell line. These cell lines were BJ-RAS, BJ-TERT and BJ-SV40. The datasets consisted of around 1000 labeled images each. For specific, biological information about the cell-lines themselves, see Section 5.6, though this is outside the scope of this thesis. A few phase-contrast images from the datasets are shown in Figure 5.9.

First, we trained models on each dataset independently, resulting in three different classifiers that were trained to identify nuclei from a single cell line. Then, we merged the three datasets into one, and trained an additional U-Net model using data from all three cell lines. The performance of each of these models were evaluated on their corresponding test sets and the "merged dataset model" was evaluated on all three test sets.

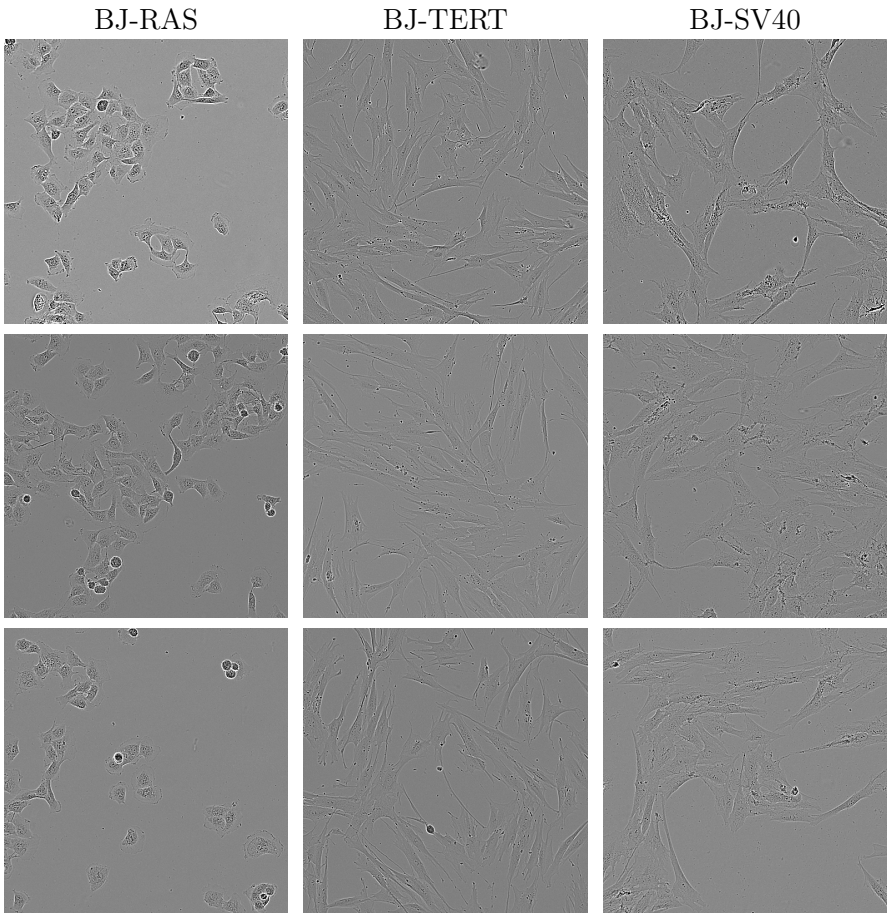

Figure 5.9: Phase-contrast images of BJ-RAS, BJ-TERT and BJ-SV40 cultured cells, taken from our datasets. All images have been processed manually for visibility.

5.6 Cell Experiment Specifics

This section contains specifics regarding the experiments that generated the data for our datasets as well as information regarding the cell lines used in this project, which may be outside the scope for this thesis. This is aimed at someone who is interested in how the cell experiments were performed and requires higher knowledge in the field of biological cell research than is expected of the reader.

5.6.1 Cell Lines

Three human fibroblast transformed cell lines (BJ) were used for live cell imaging. BJs were transformed into a tumorigenic state by targeting three genes: telomerase catalytic subunit (hTERT), SV40 large-T antigen (SV40), and H-Ras

oncoprotein (RAS). The hTERT gene allows primary human cells to multiply indefinitely. SV40 and RAS encode proteins responsible for the regulation of growth, differentiation, and survival, making the BJ-SV40 and -RAS cell lines capable of forming tumors when injected in mice [59].

5.6.2 Experiments

BJ-TERT, -SV40, and -RAS cell lines were grown in RPMI-1640 medium (31870074, ThermoFisher) supplemented with 5% fetal bovine serum (16140071, ThermoFisher) dialyzed using SnakeSkin Dialysis Tubing 3.5K MWCO (88244, ThermoFisher) and penicillin/streptomycin (15140122, ThermoFisher). Cells were seeded into 6- or 12-well cell culture plates (83.3920, SARSTEDT) and incubated for 24h to allow the cells to attach before acquiring live cell images. BioTracker™ 488 Green Nuclear Dye (SCT120, MERCK) was used to stain the nuclei for ground truth images. The BioTracker Dye (1000X) was diluted to a final concentration of 1X in cell culture medium with or without verapamil and incubated for 15 mins at 37C. Images were acquired using an IncuCyte S3 Live-Cell Analysis Instrument (SARTORIUS AG). In the methionine dropout time-lapse experiment, methionine in the culture medium was replaced with homocysteine.

Chapter 6
Results

In this chapter, I will present results from the fluorescence processing pipeline and trained U-Nets. Because our U-Net models were difficult to evaluate automatically due to the lack of ground truth, the automatic evaluation results are not to be taken as pure performance metrics. They are displayed to show how they correlate to the results that were provided by manual evaluation, so that each evaluation method can give us valuable information about the performance of our models.

6.1 Data Processing and Label Generation

6.1.1 Label Generation

A sample of fluorescence images from BJ-RAS, BJ-TERT and BJ-SV40 along with their resulting binary image labels are displayed in Figure 6.1. Raw fluorescence images are excluded since they are usually very dark and would just display as black images. In 6.1.a, we can see that the fluorescence images can be hard to analyze with the human eye, even after CLAHE has been applied. In the BJ-SV40 example, the nuclei are barely visible. To actually see how well the fluorescence images were transformed into the binary images that we used as labels, we often had to process the images manually (by using image specific settings for manual image histogram equalization, for example), resulting in images such as those in 6.1.b. The actual segmentation that was found by applying our CellProfiler pipeline to the CLAHE-treated fluorescence images in 6.1.a is shown in 6.1.c. These results are not perfect, but since we had to resort to quite generalized methods that could be able to deal with the variation that is found in the fluorescence images, errors were bound to occur.

While many labels seemed to be accurate representations of the nucleus positions marked by the fluorescence images (such as those shown in Figure 6.1), the images were sometimes poorly segmented by our image processing pipeline. Though our automatic filtering dealt with many such cases, some less extreme cases that still had significant defects remained in the datasets. Some such examples are shown in Figure 6.2.

In 6.2.a, the CLAHE-treated fluorescence images are shown. We can already see some strange effects, such as the "wave-shapes" in the BJ-RAS example. This is something that could occur when CLAHE was applied and is likely a result of the

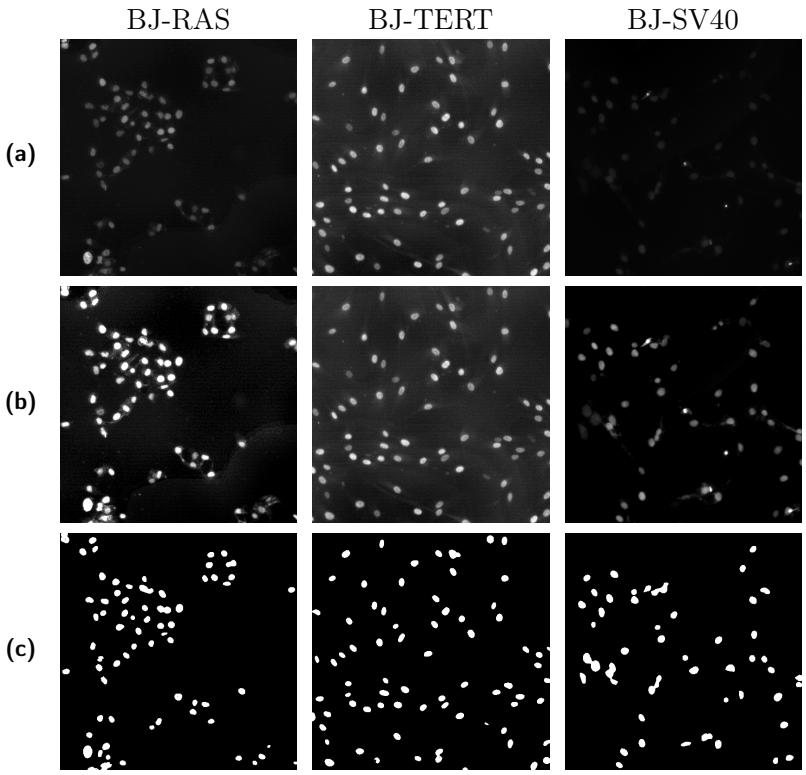

Figure 6.1: Fluorescence image samples of the three different cell lines and the resulting binary image labels (from left to right): RAS, TERT and SV40. **(a)**: CLAHE processed fluorescence images. **(b)**: Manually enhanced images in (a), exclusively for visibility in this thesis. **(c)**: Binary image labels (output from CellProfiler after processing the images in (a)).

non-constant nature of the background, which could include varying intensities of light from fluorescence or other random noise. Interestingly, by comparing it with the generated labels in 6.2.b, these artifacts seemed to have little impact on the segmentation provided by our CellProfiler pipeline. The clump at the bottom of the BJ-RAS image has a lot of "leaking fluorescence" which could be difficult for a thresholding algorithm such as Otsu's method to deal with, but it seems to include most nuclei without including much noise. The issues in the image are at the top, where nothing immediately visible is happening. By looking closely, however, a semi-circle-like shape of slightly increased brightness is present. A similar shape is present in the label, showing that our CellProfiler pipeline is sensitive to some kinds of background noise.

The BJ-TERT case is similar to the BJ-RAS case, but the BJ-SV40 case shows no large amount of segmented noise, but is missing some nuclei. A quick observation is that many nuclei are *very* similar to the background in terms of light intensity here, which of course will make segmentation based on thresholding difficult.

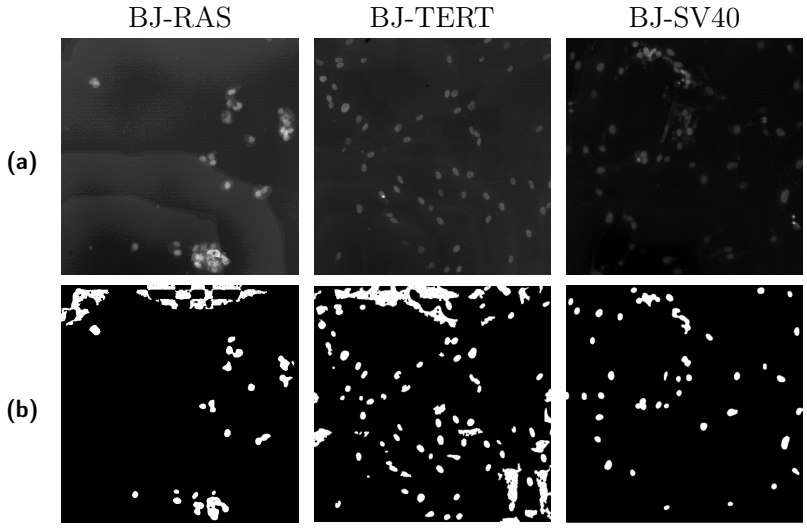

Figure 6.2: Examples of labels that have been poorly segmented from fluorescence images. **(a)**: Images from fluorescence microscopy. **(b)**: Poorly segmented binary image masks from our image processing pipeline, each showcasing major issues: segmented fluorescence "leaks", segmented noise and large amounts of missing nuclei.

6.2 Trained U-Net Performance

In this section, performance results of trained U-Net models will be presented. Sample images from each dataset are displayed in Figure 6.3. For consistency, the same images as those shown in 6.1 are used. Figure 6.3 is a sort of brief summary of our results, giving a quick glance of what the labels could look like and how the classifiers could perform on the different cell lines.

Figure 6.3.a shows one phase-contrast test case from each cell line that we trained models for and 6.3.b shows their assigned labels, made from their corresponding fluorescence images. 6.3.c displays predictions that were made by applying our trained models to the phase-contrast test cases in 6.3.a. The colorful images in 6.3.d are pixel-level comparisons between the labels in 6.3.b and the predictions in 6.3.c, like done automatic evaluation. The color-coding is green for true positive, red for false positive, black for true negative and blue for false negative. There is a lot of green present in the images, which shows a lot of promise for our method. Some chunks of false positive/negative pixels are present (especially in the BJ-SV40 example) and the exact shapes of the nuclei (across all examples) were rarely perfectly identical (which is to be expected). In the subfigures in 6.3.e we can see the predicted nuclei overlaid upon the input phase-contrast images (like in the manual evaluation method described in Section 5.4.3) which adds context to the predictions. The green pixels in these overlay images are the predicted nucleus pixels (i.e. the white pixels in the predictions in 6.3.c). Although the phase-contrast image is manually enhanced for visibility, it is still difficult to verify exact nucleus positions in the phase-contrast images, especially in the BJ-TERT and BJ-SV40 cells. For more visual representations of predictions, see appendix A, where more test cases like those in 6.3.e are shown.

For each dataset, two classifiers were evaluated: one that was trained exclusively on the corresponding dataset (referred to as pure) and one that was trained on a mix of all three datasets (referred to as mixed). In the following tables, each row contains values that were found through evaluation on a specific test set. For example, the row of "Pure Classifiers: BJ-TERT" contains the evaluation results that were obtained when testing the BJ-TERT-trained classifier on the BJ-TERT test set. Similarly, the row of "Mixed Classifier: BJ-RAS" contains the evaluation results that were obtained when testing the "mixed-cell-line"-trained classifier on the BJ-RAS test set.

6.2.1 Automatic Evaluation: Pixel-Level

Results from the automatic, pixel-level evaluation are presented in Table 6.1. These metrics were computed by comparing predictions to labels on a pixel-level. When looking at the examples in Figure 6.3, these values seem lower than expected. The accuracy is generally high, but this is likely because of the high representation of easily classified background pixels, boosting the accuracy value with a large number of true negatives. Precision, dice and IoU are often much lower than what would be expected when looking at the predictions. Recall is consistently higher than prediction. An easy conclusion to make from this is that false positives occur very frequently and while false negatives regularly appear,

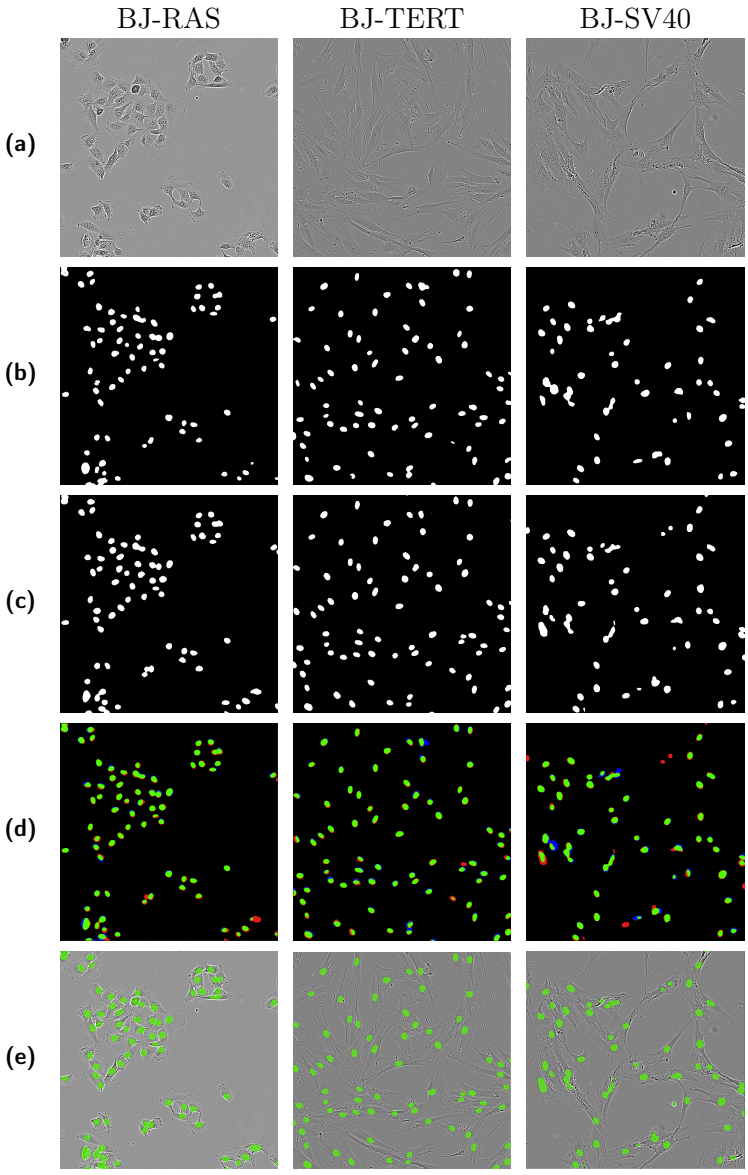

Figure 6.3: Cell images from the test sets of the three different cell lines (from left to right): BJ-RAS, BJ-TERT and BJ-SV40. **(a)**: Manually enhanced phase-contrast images. **(b)**: Binary image labels for the phase-contrast images (output from CellProfiler). **(c)**: Predicted nuclei from the phase contrast images (U-Net output). **(d)**: Labels (b) overlapped with predictions (c). Green pixels are true positives, blue pixels are false negatives, red pixels are false positives. **(e):** Predictions (c) overlaid upon the corresponding phase contrast images (a).

they are not nearly as common.

Table 6.1: Evaluation results from automatic, pixel-level evaluation. The three pure classifiers were tested on their respective test sets and the mixed classifier was tested on all three.

Automatic Evaluation Pixel-Level	Precision	Recall	Accuracy	Dice	IoU
Pure Classifiers					
BJ-RAS	0.685	0.915	0.972	0.780	0.641
BJ-TERT	0.729	0.875	0.980	0.794	0.660
BJ-SV40	0.638	0.889	0.973	0.738	0.587
Mixed Classifier					
BJ-RAS	0.727	0.894	0.976	0.798	0.666
BJ-TERT	0.736	0.874	0.980	0.798	0.666
BJ-SV40	0.683	0.990	0.973	0.744	0.594

6.2.2 Automatic Evaluation: Object-Level

Results from the automatic, object-level evaluation is presented in Table 6.2. These metrics were computed by comparing predictions to labels on an object-level (focusing on object positions rather than shapes), using the algorithm described in Section 5.6. As stated previously, this method gave us no measure of true negatives. Therefore, the accuracy metric is excluded.

The object-level scores in Table 6.2 tell an entirely different story compared to the pixel-level ones in Table 6.1. They have much higher values and we find significant increases in precision, dice and IoU. The recall is consistently over 0.98, meaning that false negative objects are rare. Since object-level is generally more interesting than pixel-level for our purposes (using predictions for cell counting and tracking), this is a reassuring find. However, since both of these evaluation methods are based on comparisons between predictions and labels, one may wonder why the values are so different. This will be discussed in Section 7.2.1.

Table 6.2: Evaluation results from automatic, object-level evaluation, using the algorithm described in Section 5.6. The three pure classifiers were tested on their respective test sets and the mixed classifier was tested on all three.

Automatic Evaluation Object-Level	Precision	Recall	Dice	IoU
Pure Classifiers				
BJ-RAS	0.944	0.986	0.964	0.930
BJ-TERT	0.920	0.982	0.950	0.904
BJ-SV40	0.770	0.990	0.869	0.768
Mixed Classifier				
BJ-RAS	0.949	0.983	0.966	0.934
BJ-TERT	0.927	0.980	0.953	0.910
BJ-SV40	0.775	0.992	0.870	0.770

6.2.3 Manual Evaluation

Results from the manual object-level evaluation by the two experts, using the method explained in Section 5.4.3, are shown in Table 6.3 and Table 6.4. The test sets for these evaluations were smaller, randomly selected subsets of the original test sets, consisting of 30 images per dataset. Since both the pure and mixed classifiers were evaluated, a total of 180 overlay images, some containing over 100 cells, were examined.

In the tables, the true positive (TP) counts are marked with an asterisk. This is because the TP count is merely an approximation of the true TP count and there are some things to keep in mind:

- We do not have a ground truth since we do not have a reliable way to count the exact number of cells in an image.
- An object (connected component) in a prediction mask can be considered a true positive unless it is marked as a false positive.
- A number of clumped cells that were connected in the prediction will be considered a single object and therefore a single true positive, given that the prediction is correct.
- Failed predictions (false positives/negatives) in clumps that were identified were still counted as false positives/negatives.
- Clumped predictions may differ when comparing the predictions of the pure and mixed classifiers. For example, a clump might be a single connected component in one prediction but split into three connected components in another. This can be a matter of a few pixels.

We based the TP count on the number of objects in the prediction mask subtracted by the number of identified false positives ($TP = object\_count - FP$). Because of clumping, the number of objects can differ between two prediction

masks even if the number of false positives (FPs) and false negatives (FNs) is the same. Thus the number of potential TPs (i.e. the number of cells in the "ground truth") is not the same for the pure and mixed classifiers and the registered TP count is an approximation. This is yet another unfortunate effect of the lack of ground truth. On the bright side, this does *not* mean that our scores are inflated. Because clumps of TPs were counted as single TPs, the true number of TPs is likely higher than our approximation ($TP_{true} \geq object\_count - FP$)

The scores in Table 6.3 and Table 6.4 appear quite extraordinary. With many metrics showing values over 0.99, it is clear that the experts agreed with the classifiers at most points. A remarkably low number of false negatives were found, and a higher (but still very low) number of false positives were found. Granted, this evaluation task is exceedingly difficult to do manually and will have some errors, so these results cannot be stated as explicit performance metrics either. Together with the values from the other evaluation methods, we can get an idea of how well our trained models perform, where they excel and where they fall short.

The manual evaluation results show no major differences between the cell lines. The differences that do exist might seem significant relative to each other, but in the grand scheme of things they are probably still within the margin of error considering how the opinions of the experts differed.

Table 6.3: Evaluation results from expert 1's manual evaluation, using the method described in Section 5.4.3.

Manual Evaluation: Expert 1	TP* count	FP count	FN count	Precision	Recall	Dice	IoU
Pure Classifiers							
BJ-RAS	2062	28	2	0.987	0.999	0.993	0.986
BJ-TERT	2143	35	7	0.984	0.997	0.990	0.981
BJ-SV40	2257	57	10	0.975	0.996	0.985	0.971
Mixed Classifier							
BJ-RAS	2099	24	4	0.989	0.998	0.993	0.987
BJ-TERT	2175	31	18	0.986	0.992	0.989	0.978
BJ-SV40	2245	48	6	0.979	0.997	0.988	0.977

Table 6.4: Evaluation results from expert 2's manual evaluation, using the method described in Section 5.4.3.

Manual Evaluation: Expert 2	TP* count	FP count	FN count	Precision	Recall	Dice	IoU
Pure Classifiers							
BJ-RAS	2071	19	0	0.991	1.000	0.995	0.991
BJ-TERT	2151	27	8	0.988	0.996	0.992	0.984
BJ-SV40	2284	30	27	0.987	0.988	0.988	0.976
Mixed Classifier							
BJ-RAS	2103	20	4	0.991	0.998	0.994	0.989
BJ-TERT	2194	12	18	0.995	0.992	0.993	0.987
BJ-SV40	2270	23	22	0.990	0.990	0.990	0.981

6.2.4 Expert Correlation

Since the values from the manual evaluation are based on opinions of two experts, we found it important to also present a representation of how much the two experts agreed with each other. Table 6.5 and Table 6.6 feature four matrices, similar to confusion matrices, that show the correlation between the evaluations. For example: the "Pure: False Positives" matrix represents what the experts found when evaluating the pure classifiers in terms of false positives. [1, 1] shows the number of times both experts found 1 false positive and has the value 13. [3, 4] shows the number of times expert 1 found 3 false positives, but expert 2 found 4 and has the value 0 (it never happened). The matrices have the 5x5 dimensions since there was no registered case of $FP > 4$ or $FN > 4$ in a single image.

Looking at the correlation Table 6.5 and Table 6.6, we can note that the experts mostly agreed when it came to false negatives and agreed often when it came to false positives. When they did disagree, it was usually by a single cell and cases where they disagreed by larger numbers were rare.

Table 6.5: Correlation tables for the manual evaluation on the pure classifiers, showing how similar the opinions of the two experts were.

Pure: FP		*Expert 2*				
		0	**1**	**2**	**3**	**4**
Expert 1	**0**	13	5	1	0	0
	1	17	13	6	1	0
	2	4	14	4	0	1
	3	0	5	2	0	0
	4	1	0	3	0	0

Pure: FN		*Expert 2*				
		0	**1**	**2**	**3**	**4**
Expert 1	**0**	57	14	3	0	0
	1	6	6	2	0	0
	2	0	1	0	0	0
	3	0	0	0	0	1
	4	0	0	0	0	0

Table 6.6: Correlation tables for the manual evaluation on the mixed classifier, showing how similar the opinions of the two experts were.

Mixed: FP		*Expert 2*				
		0	**1**	**2**	**3**	**4**
Expert 1	**0**	20	3	0	0	0
	1	21	17	4	0	0
	2	3	10	2	1	0
	3	3	2	1	0	1
	4	1	0	1	0	0

Mixed: FN		*Expert 2*				
		0	**1**	**2**	**3**	**4**
Expert 1	**0**	48	16	3	0	0
	1	6	9	4	0	0
	2	1	0	1	1	0
	3	0	1	0	0	1
	4	1	0	1	0	0

6.2.5 Overall Evaluation Results

First and foremost, the manual evaluation shows a very strong performance across all cell lines. The automatic object-level evaluation gave mostly solid scores but were not very convincing in some cases (such as the BJ-SV40 precision). The automatic pixel-level evaluation are not giving very exciting scores. The following section and the discussion will go into greater detail regarding this, but one key factor is important to keep in mind from the beginning: the scores from automatic evaluation are based on treating our labels as the ground truth, and we know that this is not fully applicable in our case.

An interesting thing when comparing the manual evaluation results with the automatic evaluation results (both object- and pixel-level) is the number of false positives. While much more common than false negatives, even in manual evaluations, the magnitude of which the number of false positives differs from the number of false negatives is completely different in manual evaluation, compared to automatic evaluation. Precision is mostly lower than recall in all our results, but in the manual evaluation results, they are quite similar and we commonly find that $\text{Recall} - \text{Precision} < 0.02$. In the automatic evaluation, this difference is often in $[0.2, 0.3]$ for pixel-level scores and in $[0.05, 0.2]$ for object-level scores. This is very likely an effect that stems from the missing nuclei in the labels. This phenomenon will be expanded on in Section 7.3.

6.2.6 Differences in Cell Lines

Some interesting phenomena could be found in specific cell lines. While comments that have been relevant for all cell lines have been made in the prior sections, a few interesting traits can be found in specific ones.

BJ-RAS

The BJ-RAS nuclei seemed to be the easiest to segment. With very high metric values from both manual evaluation and object-level evaluation, it is likely that both the label-generation pipeline and the final U-Nets worked quite well for this data.

BJ-TERT

The BJ-TERT results were very similar to the BJ-RAS results. The BJ-TERT fluorescence images had the lowest light intensity variation, making the nuclei fairly easy to segment. However, the images often included noise, which sometimes extended to the binary image labels. Since this noise is unlikely to be predicted, this is likely why the BJ-TERT values are consistently on the lower end of the three cell-lines in terms of recall in automatic evaluation.

BJ-SV40

The BJ-SV40 is unique in the way that the classifiers showed significantly worse precision on this data compared to the other cell lines. In manual evaluation though, it is not much worse than the others. A likely explanation for this is found in the fluorescence images. They were generally difficult to segment, and many labels in the BJ-SV40 datasets were missing nuclei. If the classifier still turns out good, it will predict nuclei in the test images regardless if they are missing in the labels or not, which will lead to a considerable amount of false positives in automatic evaluation.

6.2.7 Training on Mixed or Pure Cell Data

Judging by the results, there is no clear winner in terms of performance when comparing pure and mixed classifiers. This means that training classifiers on multiple cell lines is a fully viable option, which can make deployment of classifiers a lot easier. In addition, since it is possible that the mixed classifier is more regularized and has been subject to more morphologies, it is possible that it could perform well on other cell lines as well. This is something that would have to be explored and evaluated.

6.2.8 Predictions on Entries with Poor Labels

With these results in mind, it is interesting to see what predictions looked like for entries with poorly segmented labels. Since automatic evaluation was based on comparisons with the labels, a good prediction would yield poor values for our evaluation metrics if the label was a poor representation of the ground truth. In Figure 6.4, predictions (using the mixed classifier) for the entries with poorly segmented labels from Figure 6.2 are shown. 6.4.a shows the CLAHE-treated fluorescence images, 6.4.b shows the labels that were made from them, 6.4.c shows the predictions that were made by running predictions on the corresponding phase-contrast images. Finally, said phase-contrast images are displayed in 6.4.d.

Figure 6.4 shows that the predictions here are far better representations of the ground truth than the labels. Cases like these likely play a key part in the lower automatic evaluation scores we see in Table 6.1 and Table 6.2. This is further supported by the fact that the pixel-level scores are so much worse than the object-level scores: artifacts like the ones we see in the faulty BJ-TERT label (6.4.b) cover many pixels but are just a few connected components. Thus they provide *a lot* of misinformation on pixel-level, but much less on an object-level.

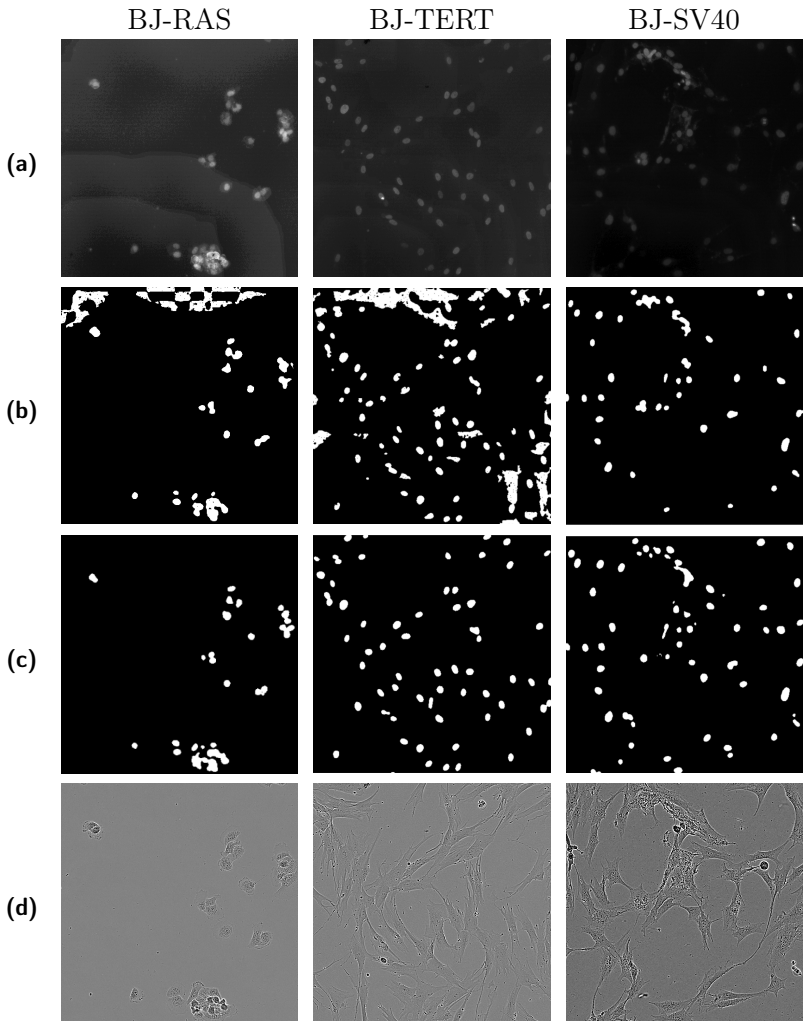

Figure 6.4: An expansion of Figure 6.2, adding the input images as well as predictions on them. **(a)**: Fluorescence images. **(b)**: Labels, made from the fluorescence images in (a). **(c)**: Predicted nucleus positions. **(d)**: The input phase-contrast images.

To investigate this further, the automatic evaluation scores for the "bad label" images in 6.4 are presented in Table 6.7. The large amounts of segmented noise in the BJ-RAS and BJ-TERT examples had a massive impact on the pixel-level recall due to the large amount of supposed false negatives, which then extended to very low pixel-level dice and IoU scores. As suspected, the object-level scores took a hit too, but to a much smaller extent. Precision seemed somewhat consistent with what we were used to. Nevertheless, it is clear that cases like these had an impact on the overall scores.

BJ-SV40 sticks out here, because unlike BJ-RAS and BJ-TERT, the scores

are actually somewhat similar to the average scores. After reviewing the datasets, we found that the BJ-SV40 labels commonly lacked some nuclei, and this finding further supports this. Since the trained classifiers were predicting BJ-SV40 nuclei quite well according to manual evaluation (Table 6.3 and Table 6.4), this further explains the poor precision, dice and IoU scores that we found through automatic pixel-level and object-level evaluation.

In all cases, the pixel-level accuracy remained high, even though most other scores were significantly lower than average. This further shows how the accuracy metric is influenced by the very high number of correctly classified background pixels.

Table 6.7: Automatic evaluation scores for the test cases with very noisy labels that were presented in Figure 6.4.

Automatic Evaluation: Very Noisy Labels	Precision	Recall	Accuracy	Dice	IoU
Pixel-Level					
BJ-RAS	0.775	0.320	0.959	0.453	0.293
BJ-TERT	0.778	0.296	0.916	0.428	0.273
BJ-SV40	0.609	0.822	0.979	0.700	0.538
Object-Level					
BJ-RAS	0.880	0.710	-	0.786	0.647
BJ-TERT	0.967	0.656	-	0.781	0.641
BJ-SV40	0.698	0.978	-	0.815	0.688

Chapter 7
Discussion and Conclusion

In this chapter, I will discuss our methodology, our results and compare our project with some related work.

7.1 Automatic Label Generation

One of the most interesting and important parts of this project is the automatic generation of labels through fluorescence microscopy, image analysis and traditional segmentation approaches.

7.1.1 Robustness

Our results show that some of the fluorescence images were processed really well, while some had missing nuclei or even noise segmented as nuclei. The same pipeline could be used for several cell lines but fell short when attempting to process images of cells that were partly rejecting the nuclear dye. But even when the fluorescence images were segmented perfectly, there was still no guarantee that the resulting binary label was a perfect representation of the nuclei, since it was always entirely possible that some cells did not pick up or respond to the nuclear dye. Due to this, there was always some uncertainty and the automatically generated labels were often imperfect.

An important observation to be made here is that our labels are *not* a ground truth, which has interesting implications since most supervised learning approaches use these terms synonymously.

7.1.2 Impact of Imperfect Labels

Despite these imperfect labels, our method shows promising results. While it is hard to know the exact implications of our imperfect labels, we can find that background and cytoplasm is rarely classified as nuclei, even though they are sometimes labeled as nuclei in training data. The keyword for this behaviour might be *randomness*.

When looking for patterns in our training data, most nuclei are segmented as nuclei and background is segmented as background. The deviations from these patterns derive from amplified background noise and individual cells that have not

reacted to nuclear dye. Both of these deviations occur seemingly *randomly*: there are no clear patterns in the phase-contrast images that lead to these deviations. This randomness may lead to these deviations having minimal effects on the training. The clear patterns will consistently push the training in right direction, while the deviations will attempt to nudge the training into some random direction that likely won't be supported by other deviations. Thus the clear patterns will be the "overwhelming force". It is possible that this random noise could even act as a form of regularization since it pushes the training "against the current" which could reduce overfitting. This, however, would need to be further investigated.

7.1.3 Comparison with Alternatives to Our Label Generation

Manually Annotated Data

While manual labeling by experts is often viewed as the most accurate, it is also very time consuming and is not feasible when working with large amounts of data, especially with data like microscopy images where labeling a single image can take a long time. Simultaneously, there is no real guarantee that manually labeled data is correctly labeled since it is entirely based on some person's work. Especially when the labeling is difficult even for humans and experts may disagree with each other (like in our manual evaluation).

Purely Synthetic Data

A way to completely circumvent these issues, guaranteeing that all data is correctly labeled without manual annotation is to use purely synthetic data, i.e data that is completely synthesized by your own code. If the objects in an image are generated by your own program, you always know the positions of objects and labeling becomes trivial. The obvious issue with synthetic data is that it may be a poor approximation to real data and it may be hard to know how a model that has been trained on synthetic data will perform on real data. If the synthetic data is based on good simulations this data may be quite accurate, but if it looks nothing like the real data that you want to process there is little point in doing this.

7.2 U-Net Performance

Our evaluation methods gave us three types of evaluation results: automatic pixel-level comparisons with labels, automatic object-level comparisons with labels and manual object-level comparisons with the phase-contrast images. By reviewing our results, our primary goal was, of course, to document the performance of the trained models and a secondary goal was to see what kind of information the different evaluation methods could give us.

7.2.1 Differences in Evaluation Results

A large point of interest is the disconnect that we find between the scores from the different evaluation methods. The manual object-level evaluation showed excellent

results, but the automatic pixel-level evaluation left much to be desired. The automatic object-level evaluation gave us much higher scores than the automatic pixel-level evaluation did in many cases, even though both methods were based on comparisons between predictions and labels. It is understandable that the manual and automatic evaluation methods can give different results, but if the prediction is very similar to the label on an object level it might feel strange that the pixel-level scores are so low.

Something that can be seen in many test cases (such as the ones presented in Figure 6.3), is that even though true positives are common and the predictions seem accurate, the shapes of the nuclei are essentially never identical to the ones in the labels. In fact, by magnifying an area in one of the images, we will find a number of false positives and false negatives, as shown in Figure 7.1.

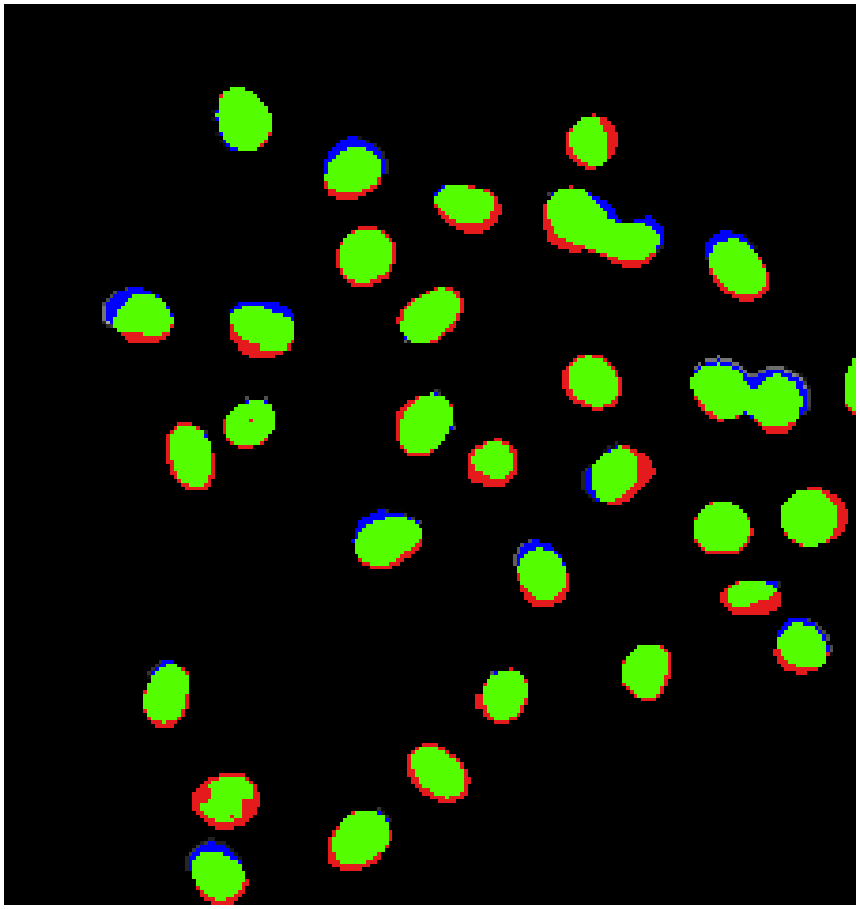

Figure 7.1: A magnification of a prediction-label overlap from Figure 6.3.d. Green pixels represent true positives, red pixels represent false positives and blue pixels represent false negatives.

These small errors add up quickly and have a significant impact on the pixel-

level evaluation while simultaneously having no impact on the object-level evaluation. If we consider that the nuclei in fluorescence images act as position markers rather than exact shapes of the objects we are trying to find, we can be quite happy if we get good results from the object-level evaluation even if the values from the pixel-level evaluation are lacking.

7.2.2 The α Hyperparameter

All that the α hyperparameter in the binary focal cross entropy function really does is penalize miss-classifications of one class harder than the other. Since the γ hyperparameter already ensures that harder classifications are weighted much more heavily than easier classifications, the α hyperparameter applies a feature of varying usefulness: if the classifier is uncertain, it will prefer to predict the class that is down-weighted by α. This is because the added loss from miss-classifying the down-weighted class is much lower than the added loss from miss-classifying the up-weighted class.

To show this, we trained three models on the mixed dataset with the author's suggested values for α ($\alpha = 0.25$ and its counterpart $\alpha = 0.75$) [37]. When $\alpha = 0.25$, false positives contribute much more to the overall loss than false negatives, and so, the classifier will opt to predicting negatives when unsure. $\alpha = 0.75$ gives the opposite, so that false negatives contribute much more to the overall loss than false positives. Sample predictions are shown in Figure 7.2.

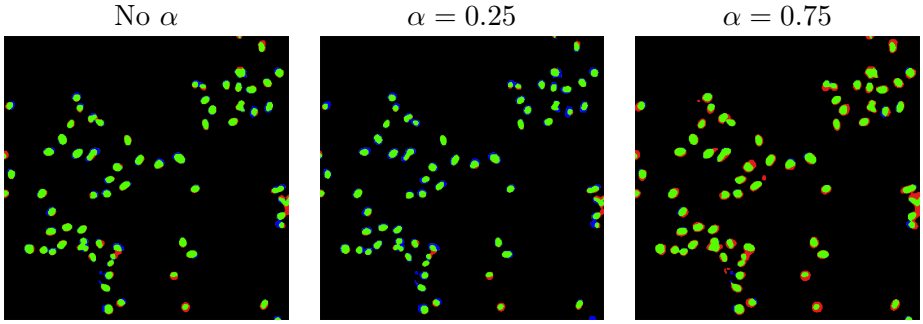

Figure 7.2: As before: green = true positive, black = true negative, red = false positive and blue = false negative. Note that the predicted nuclei in $\alpha = 0.75$ are significantly larger than in the other images and are often surrounded by false positive pixels. Also, some false positive objects appear. Conversely, the predicted nuclei in $\alpha = 0.25$ are smaller and surrounded by false negatives. Also, the nucleus in the bottom right is miss-classified as negative.

Using α for weighting can be useful but from our limited evaluation we found no major impact on performance that would warrant doing so by default on our data. As explained before, the γ term already adds class weighting indirectly by

lowering the impact of easy classifications on the loss function, which encompasses the vast majority of background pixels.

7.3 Credibility of Evaluation

7.3.1 Automatic Evaluation

Traditional evaluation by comparing predictions with known labels gives some information, but knowing that our labels are not a ground truth, we cannot use this method for absolute performance metrics. This is especially evident in cases where our models yield predictions that are more accurate than the provided label. Such a case is visualized in Figure 7.3 and the images are from the BJ-SV40 sample in Figure 6.1 and Figure 6.3.

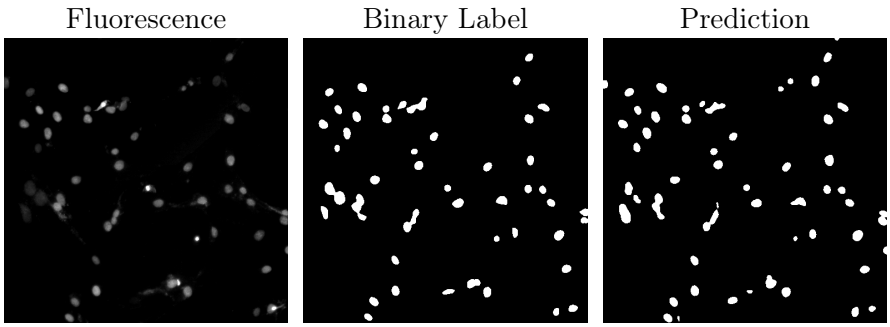

Figure 7.3: An example of a prediction that contains nuclei that were not segmented correctly in the label.

In this sample, we can see how two nuclei in the top right were incorrectly segmented when comparing the fluorescence image with its resulting binary label. The U-Net model, however, finds the nuclei and includes them in the prediction. This is obviously a good thing, but when we compute scores that are based on differences between the label and the prediction, this behaviour will lower the score.

In addition, we find a predicted nucleus in the bottom center that is *barely* visible in the fluorescence image. As stated before, some nuclei don't even show clearly in the fluorescence images, so in order to truly know whether this is a false positive or not, we have to compare the prediction to the input phase-contrast image.

That being said, the automatic evaluation metrics can give some overview of the performance, showing that the model is at least moving in the right direction and not just guessing randomly. However, since "false positives" that are in fact not false (but are named as such due to false negatives in the label) are not that rare, metrics that have a large focus on false positives (such as precision) cannot be trusted.

7.3.2 Manual Evaluation

While manual evaluation theoretically gives very good information, it not only requires a very experienced evaluator but also takes a lot of time. Because of this, we are forced to work with subsets of the test data, which means that we base our evaluation on less samples than what we would do with the automatic evaluation. Although, while this is a downside, the fact that the information that is provided by manual evaluation is much more representative of the actual performance, it is likely that manual evaluation is much more reliable than automatic evaluation here.

7.4 Related Work

Before concluding the chapter, I will present some related work and compare our project with theirs.

7.4.1 In Silico Labeling: Predicting Fluorescent Labels in Unlabeled Images - Christiansen et al.

In this project, the authors trained deep learning models for predicting *fluorescent labels* in transmitted-light microscopy images (e.g. phase-contrast) [60]. In other words, they trained models that would input a transmitted-light microscopy image and output a synthetic fluorescence microscopy image. This, by extension, could then be used to detect various features in those images, such as cell locations, cell types, cell health and types of subcellular structures. They used data that was provided by several laboratories, consisting of transmitted-light microscopy images and fluorescence microscopy images [60].

While their models could perform cell identification (and much more for that matter), they tackled a problem at a grander scale, using more detailed data and a more complex network. This, of course, comes at a cost: the authors report that their training process took roughly 2 weeks and predictions on 1024x1024 images took about 256 seconds [60]. In comparison, our models (while of course being trained to solve a simpler task) took 5-15 hours to train and would take about 0.12 seconds to predict nuclei in a 1024x1024 image. While the training period might be a very minor issue if training new networks is of limited interest, this difference in prediction times is very important when running predictions on large time-lapse datasets that can consist of over 20,000 images. For 1024x1024 images, such a time-lapse would take approximately 5,120,000 seconds (about 2 months) to predict using their model and approximately 2,400 seconds (about 40 minutes) to predict using our model. For a group that runs microscopy experiments frequently, this run-time will add up and be of great importance.

7.4.2 Label-free prediction of three-dimensional fluorescence images from transmitted-light microscopy - Ounkomol et al.

Similar to the project by Christiansen et al., the authors behind this paper trained deep learning models to predict fluorescence from transmitted-light microscopy

[61]. The models were trained using transmitted-light images with corresponding fluorescence images as labels [61].

7.4.3 Practical segmentation of nuclei in brightfield cell images with neural networks trained on fluorescently labelled samples - Fishman et al.

This project is very similar to ours. The authors focused on using U-Net models (as well as some others) to segment nuclei in brightfield (transmitted-light microscopy, similar to phase-contrast) cell images [62]. Their data was handed to them, so they did not conduct their own cell experiments for training data. They generated labels semi-automatically by processing fluorescence images using a software that ran some segmentation algorithm. They evaluated and adjusted the resulting binary image manually before including them as training data. They trained models for different cell lines, using "pure" datasets consisting of cell images from a single cell line, as well as "mixed" datasets consisting of cell images from multiple cell lines [62].

All in all, while this project had a large focus on evaluating different models, the work was conceptually very similar to ours. The purposes for the projects were quite different however, since ours was more focused on building a functional pipeline for building and using deep learning models for data extraction in cell experiments. In addition, we made having a fully automatic process that works on in-house generated data a priority.

7.5 Conclusion

While work still remains, our method shows great promise for automatic cell identification. Using this method, it is possible to train and use U-Net models for cell identification using in-house generated, automatically labeled data and obtain accurate results. Despite having an automatic pixel-level evaluation that showed mediocre results, automatic object-level dice scores of over 0.95 show great promise and manual object-level dice scores of around 0.99 support that the predictions are on par with expert annotation. It is even possible to train classifiers on multiple cell lines, making practical use even easier. Knowing that the implementation can likely be improved by using more recently developed models as well as a more versatile segmentation pipeline for the fluorescence images, the cell identification can likely improve. While automatic cell identification is valuable on its own, solving local issues with cell-tracking algorithms would provide researchers with more high-content data from their experiments. Our hope is that this, along with similar projects, can be a helping hand in solving the methionine dependency problem as well as problems that are studied with microscopy time-lapses.

Chapter 8
Future Work

Here, in the final chapter of the thesis, I will point out some interesting points of future work that could expand upon what we found in this project.

8.1 Cell Tracking

First and foremost, successfully linking our cell identification models to cell tracking algorithms would provide researchers with very useful data from their experiments. This was originally a goal for this project, but after finding a number of issues (many which originated from technical specifications in the microscope) when attempting to apply the tracking algorithms, we had to cut it due to time restrictions. Unfortunately, working around these turned out to be more work than anticipated.

Our plan was to apply our U-Nets to a microscopy time-lapse to get solid cell segmentation and then use the resulting segmented time-lapse with the cell-tracking methods in Baxter Algorithms (BA). BA was chosen since it was not only amongst the top performing cell tracking methods available [9], but was also developed under Joakim Jaldén's supervision, who assisted us in this project. Unfortunately, when attempting to use BA, we quickly found some issues that needed resolving. The severity of these issues were unclear and would need to be evaluated in future work. Three seemingly harmful problems were found:

Camera Shifts

A technical issue that arose with the microscope that we used was camera shifts. In our microscope, the wells are divided into smaller parts called *tiles* and the microscope moves around the well, imaging one tile at a time. When imaging tiles at different time points, the images were not always taken from the exact same point. This would result in a shift, where all cells would move slightly to another direction. Some cells would exit the image and some cells would appear from seemingly nowhere. Then, if the image shifted back, the lost cells would reappear and the newly introduced cells would be gone. This was troublesome for the tracking algorithms. This could potentially be solved by post-processing the predictions, or by using some stabilizing point in the wells. Of course, it is not a problem if you do not use a moving microscope.

Cells Entering and Exiting a Tile

Since we used a tiled well, cells would sometimes enter or exit the tile. This was also be troublesome for the tracking, since cells randomly appear and disappear from the time-lapse. Furthermore, individual cells that disappear and then reappear later would count as new cells, even though its "lifespan" had already started at an earlier time-point.

Colliding Nuclei

While this issue is not nearly as prominent as it could be, thanks to us segmenting nuclei rather than entire cells, it was still an evident problem. When nuclei collide, the segmentation shows them as a single cell. When they separate later on, it is hard to know which nucleus is which and whether a division has occurred or not. While there are methods for separating colliding objects (such as a watershed transform), such methods might do harm as well as good and would need to be evaluated.

Given more time, we would like to explore workarounds for these issues further as well as evaluate how harmful their effects really are, so that we could successfully apply BA for cell-tracking and take this project to the next level.

8.2 Improving the Automatic Label Generation

While a simple approach seemed good for this, it is very likely that this step could be improved. Having a more robust algorithm for this would increase the quality of the labels and reduce the need for filtering, allowing larger datasets to be extracted from cell experiments.

It is possible that different approaches would be suitable for different cell lines. While taking this into account could increase the amount of work that would be required for preparing training data, it would be interesting to see if some collection of algorithms could be applied to reoccurring data.

It is also possible that a different staining method could yield images that are more easily segmented. In our project, we were limited by our microscope. In another experiment, it would be interesting to view the effects of other nuclear stains, such as DAPI [63].

8.3 Utilizing other Deep Learning Models

Since the U-Net's conception in 2015, several new versions of it, as well as different architectures all together, have been proposed [10]. While the original U-Net has proven sturdy for this kind of task, exploring how different architectures would adapt to this setting would be very interesting.

Hyperparameter tuning for the standard U-Net would also be interesting to look at. Though with the current situation featuring noisy labels, it is unclear if a hyperparameter sweep (such as WandB [64]) would be useful.

8.4 Exploring other Learning Techniques

While we used a fully supervised learning approach, it is possible that semi-supervised learning would be a better fit for our task. By using a lower number of labeled images for the learning, we could ensure that we only include high-quality labels in our learning while still providing large datasets. There have been numerous cases where a semi-supervised learning based U-Net has been explored for image segmentation in the biomedical field, and a recurring reason why is this lack of labeled data [65, 66]. While there is no guarantee that this would improve the results, it would be interesting to combine our exploration of generating data with an exploration of training methods.

8.5 Review and Improve Data Augmentation

In this project, simple methods were applied for data augmentation. The effects of these could be reviewed in order to find what type of augmentations are especially useful and emphasize those. In addition, there are some very interesting augmentation methods out there today [67] which would certainly be worth exploring. This could be especially useful when training classifiers on multiple cell lines or trying to train classifiers for new cell experiments, as the regularizing powers of data augmentation may improve flexibility for identifying foreign cell morphologies that were not present in the training data.

8.6 Explore Cell Counting Methods

Automatic cell counting is one of the main applications for our method. During the project we resorted to counting cells by counting connected components in the predicted masks. This method comes with some major drawbacks though, since clumped cells count as single objects and if some small part of the background was misclassified as a nucleus, it could count as a connected component even if it was just a few pixels. We considered an alternative, area based counting method, where we would base our cell count on the total area of segmented nuclei in the prediction. Since the sizes of nuclei are fairly consistent, we could divide the total segmented area by the known area of an average nucleus to find an approximation for the number of cells in the image. This would solve the clumping issue to a large extent, and connected components that only consist of a few pixels would not have a large impact on the cell count.

8.7 Explore More Cell Lines

It would be interesting to deploy the current classifiers on other cell lines and see how they do. It would also be interesting to accumulate data from many other cell lines and see if it would be possible to train a very general nucleus segmentation classifier. If so, that model could easily be deployed to research groups, removing the need for training entirely.

8.8 Combining Classifiers

While our models could do quite well for cell identification, there may be specific things that could be interesting to detect, such as impending division and death. By training classifiers to detect these things specifically, we could combine them with general cell identification classifiers to obtain a lot of information without even applying tracking algorithms. This is a project on its own though, since it would require labeled data for said events.

8.9 Transfer Learning

We trained our U-Net models using a He Normal weight initialization strategy, as suggested by the authors of the original U-Net paper [1]. We could, however, explore using weight initialization based on prior trained models, as seen in some papers featuring the U-Net for biomedical image segmentation [68, 69]. As an extension of this, it would also be interesting to see how feasible it would be to re-train an existing model of ours to predict nuclei in another cell line. If this form of transfer learning is a simple task, it would benefit researchers greatly, knowing that they could perform smaller re-training sessions to obtain new models, instead of doing full-scale training from scratch.

8.10 Deployment

It is important that our software is easy to use if we want researchers to make use of it. This is especially important since the main target audience are biologists that may not be overly familiar with code structures. To do this, we want to package the code in a way that it can be installed and run without any hassle. For this to work, we would ideally get rid of the ImageJ/CellProfiler steps in our pipeline and "translate" those parts into python code (which could probably largely be done through available computer vision packages such as OpenCV). In addition, some work is required for making the package and scripts as user-friendly as possible.

References

[1] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. 2015.

[2] Paul Cavuoto and Michael F. Fenech. A review of methionine dependency and the role of methionine restriction in cancer growth control and life-span extension. *Cancer Treatment Reviews*, 38(6):726–736, October 2012.

[3] J O Mecham, D Rowitch, C Douglas Wallace, Peter H Stern, and Robert M Hoffman. The metabolic defect of methionine dependence occurs frequently in human tumor cell lines. *Biochemical and biophysical research communications*, 117(2):429–434, 1983.

[4] Anusua Basu, Pradip Senapati, Mainak Deb, Rebika Rai, and Krishna Gopal Dhal. A survey on recent trends in deep learning for nucleus segmentation from histopathology images. *Evolving Systems: An Interdisciplinary Journal for Advanced Science and Technology*, pages 1 – 46, 2023.

[5] Yi Gu, Aiguo Chen, Xin Zhang, Chao Fan, Kang Li, and Jinsong Shen. Deep learning based cell classification in imaging flow cytometer. *ASP Transactions on Pattern Recognition and Intelligent Systems*, 1(2):18–27, Jun. 2021.

[6] Partha Pratim Mondal, Alberto Diaspro, and service) SpringerLink (Online. *Fundamentals of Fluorescence Microscopy. [Elektronisk resurs] Exploring Life with Light.* Springer Netherlands, 2014.

[7] Valentin Magidson and Alexey Khodjakov. Chapter 23 - circumventing photodamage in live-cell microscopy. *Methods in Cell Biology*, 114:545 – 560, 2013.

[8] (Ed.) Simon Zabler. *Phase-Contrast and Dark-Field Imaging. [Elektronisk resurs].* MDPI (Multidisciplinary Digital Publishing Institute, 2019.

[9] Vladimír Ulman, Martin Maška, Klas E G Magnusson, Olaf Ronneberger, Carsten Haubold, Nathalie Harder, Pavel Matula, Petr Matula, David Svoboda, Miroslav Radojevic, Ihor Smal, Karl Rohr, Joakim Jaldén, Helen M Blau, Oleh Dzyubachyk, Boudewijn Lelieveldt, Pengdong Xiao, Yuexiang Li, Siu-Yeung Cho, Alexandre C Dufour, Jean-Christophe Olivo-Marin, Constantino C Reyes-Aldasoro, Jose A Solis-Lemus, Robert Bensch, Thomas Brox, Johannes Stegmaier, Ralf Mikut, Steffen Wolf, Fred A Hamprecht,

Tiago Esteves, Pedro Quelhas, Ömer Demirel, Lars Malmström, Florian Jug, Pavel Tomancak, Erik Meijering, Arrate Muñoz-Barrutia, Michal Kozubek, and Carlos Ortiz-de Solorzano. An objective comparison of cell-tracking algorithms. *Nature Methods: Techniques for life scientists and chemists*, 14(12):1141 – 1152, 2017.

[10] J. (1) Kugelman, J. (1) Allman, S.A. (1) Read, S.J. (1) Vincent, M.J. (1) Collins, 4) Alonso-Caneiro, D. (1, 3) Tong, J. (2, 3) Kalloniatis, M. (2, and 5 6) Chen, F.K. (4. A comparison of deep learning u-net architectures for posterior segment oct retinal layer segmentation. *Scientific Reports*, 12(1), 2022.

[11] Lucidchart. `https://www.lucidchart.com/`.

[12] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[13] service) SpringerLink (Online and service) SpringerLink (Online. *Applications of Artificial Intelligence and Machine Learning. [Elektronisk resurs] : Select Proceedings of ICAAAIML 2020.* Lecture Notes in Electrical Engineering: 778. Springer Nature Singapore, 2021.

[14] Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, and Thomas B. Schön. *Machine Learning - A First Course for Engineers and Scientists.* Cambridge University Press, 2022.

[15] Jung Alexander. *Machine Learning: The Basics. [Elektronisk resurs].* Springer, 2022.

[16] Gérard Dreyfus. *Neural Networks. Methodology and Applications.* Springer Berlin Heidelberg, 2005.

[17] *Image Segmentation. [Elektronisk resurs].* InTech, 2011.

[18] Y. Prajna, M.K. Nath, D. Harvey, H. Kar, S. Verma, and V. Bhadauria. A survey of semantic segmentation on biomedical images using deep learning. National Institute of Technology Puducherry, Karaikal, India, 2021.

[19] *Artificial Neural Networks. [Elektronisk resurs].* Springer US, 2021.

[20] *The handbook of brain theory and neural networks.* A Bradford book. MIT, 2003.

[21] D. Livingstone. *Artificial neural networks. methods and applications.* Methods in molecular biology: 458. Humana Press, 2008.

[22] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[23] A.C.C. Liu and O.M.K. Law. *Artificial Intelligence Hardware Design: Challenges and Solutions.* Wiley, 2021.

[24] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. 2021.

[25] Sebastian Ruder. An overview of gradient descent optimization algorithms. 2016.

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

[27] Jason D Lee, Max Simchowitz, Michael I Jordan, and Benjamin Recht. Gradient descent only converges to minimizers. In *Conference on learning theory*, pages 1246–1257. PMLR, 2016.

[28] Simon S Du, Chi Jin, Jason D Lee, Michael I Jordan, Aarti Singh, and Barnabas Poczos. Gradient descent can take exponential time to escape saddle points. *Advances in neural information processing systems*, 30, 2017.

[29] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

[30] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: where bigger models and more data hurt*. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12):124003, dec 2021.

[31] Dipanwita Sinha Mukherjee and Naveen Gururaja Yeri. Investigation of weight initialization using fibonacci sequence on the performance of neural networks*. *2021 IEEE Pune Section International Conference (PuneCon), Pune Section International Conference (PuneCon), 2021 IEEE*, pages 1 – 8, 2021.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. 2015.

[33] Zari Farhadi, Hossein Bevrani, and Mohammad-Reza Feizi-Derakhshi. Combining regularization and dropout techniques for deep convolutional neural network. *2022 Global Energy Conference (GEC), Global Energy Conference (GEC), 2022*, pages 335 – 339, 2022.

[34] Sungheon Park and Nojun Kwak. *Analysis on the Dropout Effect in Convolutional Neural Networks.*, volume 10112 of *Lecture Notes in Computer Science. 10112*. Springer International Publishing, 2017.

[35] Chi Jin, Praneeth Netrapalli, and Michael I Jordan. Accelerated gradient descent escapes saddle points faster than gradient descent. In *Conference On Learning Theory*, pages 1042–1085. PMLR, 2018.

[36] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014.

[37] T. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar. Focal loss for dense object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence, Pattern Analysis and Machine Intelligence, IEEE Transactions on, IEEE Trans. Pattern Anal. Mach. Intell*, 42(2):318 – 327, 2020.

[38] Suárez-Paniagua Víctor and Segura-Bedmar Isabel. Evaluation of pooling operations in convolutional architectures for drug-drug interaction extraction. *BMC Bioinformatics*, 19(S8):39 – 47, 2018.

[39] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 2014.

[40] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. 2014.

[41] M.D. Zeiler, G.W. Taylor, and R. Fergus. Adaptive deconvolutional networks for mid and high level feature learning. *2011 International Conference on Computer Vision, Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2018 – 2025, 2011.

[42] Huimin Huang, Lanfen Lin, Ruofeng Tong, Hongjie Hu, Qiaowei Zhang, Yutaro Iwamoto, Xianhua Han, Yen-Wei Chen, and Jian Wu. Unet 3+: A full-scale connected unet for medical image segmentation. *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Acoustics, Speech and Signal Processing (ICASSP), ICASSP 2020 - 2020 IEEE International Conference on*, pages 1055 – 1059, 2020.

[43] Jeroen Bertels, Tom Eelbode, Maxim Berman, Dirk Vandermeulen, Frederik Maes, Raf Bisschops, and Matthew B. Blaschko. *Optimizing the Dice Score and Jaccard Index for Medical Image Segmentation: Theory and Practice.*, volume 11765 of *Lecture Notes in Computer Science. 11765*. Springer International Publishing, 2019.

[44] Vicar Tomas, Balvan Jan, Jaros Josef, Jug Florian, Kolar Radim, Masarik Michal, and Gumulec Jaromir. Cell segmentation methods for label-free contrast microscopy: review and comprehensive comparison. *BMC Bioinformatics*, 20(1):1 – 25, 2019.

[45] Versha Thakur and Harjinder Singh. An artificial neural network based adaptive histogram equalization algorithm for enhancement of low contrast images. *2021 8th International Conference on Computing for Sustainable Global Development (INDIACom), Computing for Sustainable Global Development (INDIACom), 2021 8th International Conference on*, pages 268 – 273, 2021.

[46] 2) Reza, A.M. (1. Realization of the contrast limited adaptive histogram equalization (clahe) for real-time image enhancement. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 38(1):35–44 – 44, 2004.

[47] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979.

[48] Agnieszka Mikolajczyk and Michal Grochowski. Data augmentation for improving deep learning in image classification problem. *2018 International Interdisciplinary PhD Workshop (IIPhDW), Interdisciplinary PhD Workshop (IIPhDW), 2018 International*, pages 117 – 122, 2018.

[49] Raschka Sebastian and Mirjalili Vahid. *Python Machine Learning : Machine Learning and Deep Learning with Python, Scikit-learn, and TensorFlow 2, 3rd Edition.* Packt Publishing, 2019.

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

[51] François Chollet et al. Keras. `https://keras.io`, 2015.

[52] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[53] Anaconda software distribution, 2020.

[54] conda-forge community. The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem, July 2015.

[55] Johannes Schindelin, Ignacio Arganda-Carreras, Erwin Frise, Verena Kaynig, Mark Longair, Tobias Pietzsch, Stephan Preibisch, Curtis Rueden, Stephan Saalfeld, Benjamin Schmid, Jean-Yves Tinevez, Daniel James White, Volker Hartenstein, Kevin Eliceiri, Pavel Tomancak, and Albert Cardona. Fiji: an open-source platform for biological-image analysis. *Nature Methods*, 9(7):676 – 682, 2012.

[56] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. Nih image to imagej: 25 years of image analysis. *Nature Methods*, 9(7):671 – 675, 2012.

[57] D.R. (1) Stirling, A.M. (1) Lucas, A.E. (1) Carpenter, B.A. (1) Cimini, A. (1) Goodman, and M.J. (2) Swain-Bowden. Cellprofiler 4: improvements in speed, utility and usability. *BMC Bioinformatics*, 22(1), 2021.

[58] C3SE. `https://www.c3se.chalmers.se/`.

[59] William C. Hahn, Christopher M. Counter, Ante S. Lundberg, Roderick L. Beijersbergen, Mary W. Brooks, and Robert A. Weinberg. Creation of human tumour cells with defined genetic elements. *Nature*, 400(6743):464, 1999.

[60] Eric M. Christiansen, Samuel J. Yang, D. Michael Ando, Ashkan Javaherian, Gaia Skibinski, Scott Lipnick, Elliot Mount, Alison O'Neil, Kevan Shah, Alicia K. Lee, Piyush Goyal, William Fedus, Ryan Poplin, Andre Esteva, Marc Berndl, Lee L. Rubin, Philip Nelson, and Steven Finkbeiner. In silico labeling: Predicting fluorescent labels in unlabeled images. *Cell*, 173(3):792 – 803, 2018.

[61] Chawin Ounkomol, Sharmishtaa Seshamani, Mary M. Maleckar, Forrest Collman, and Gregory R. Johnson. Label-free prediction of three-dimensional fluorescence images from transmitted-light microscopy. *Nature Methods: Techniques for life scientists and chemists*, 15(11):917 – 920, 2018.

[62] Chawin Ounkomol, Sharmishtaa Seshamani, Mary M. Maleckar, Forrest Collman, and Gregory R. Johnson. Label-free prediction of three-dimensional fluorescence images from transmitted-light microscopy. *Nature Methods: Techniques for life scientists and chemists*, 15(11):917 – 920, 2018.

[63] M. Rieckher, A.F.C. Lopes, and B. Schumacher. Chapter 11 - genome stability in caenorhabditis elegans. In Igor Kovalchuk and Olga Kovalchuk, editors, *Genome Stability*, pages 163–186. Academic Press, Boston, 2016.

[64] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[65] Gerda Bortsova, Florian Dubost, Laurens Hogeweg, Ioannis Katramados, and Marleen De Bruijne. Semi-supervised medical image segmentation via learning consistency under transformations. In *Medical Image Computing and Computer Assisted Intervention–MICCAI 2019: 22nd International Conference, Shenzhen, China, October 13–17, 2019, Proceedings, Part VI 22*, pages 810–818. Springer, 2019.

[66] Rushi Jiao, Yichi Zhang, Le Ding, Rong Cai, and Jicong Zhang. Learning with limited annotations: a survey on deep semi-supervised learning for medical image segmentation. *arXiv preprint arXiv:2207.14191*, 2022.

[67] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[68] Dorothy Cheng and Edmund Y. Lam. Transfer learning u-net deep learning for lung ultrasound segmentation, 2021.

[69] Anindya Apriliyanti Pravitasari, Nur Iriawan, Mawanda Almuhayar, Taufik Azmi, Irhamah Irhamah, Kartika Fithriasari, Santi Wulan Purnami, and Widiana Ferriastuti. Unet-vgg16 with transfer learning for mri-based brain tumor segmentation. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, 18(3):1310–1318, 2020.

Appendix A
Prediction Overlays

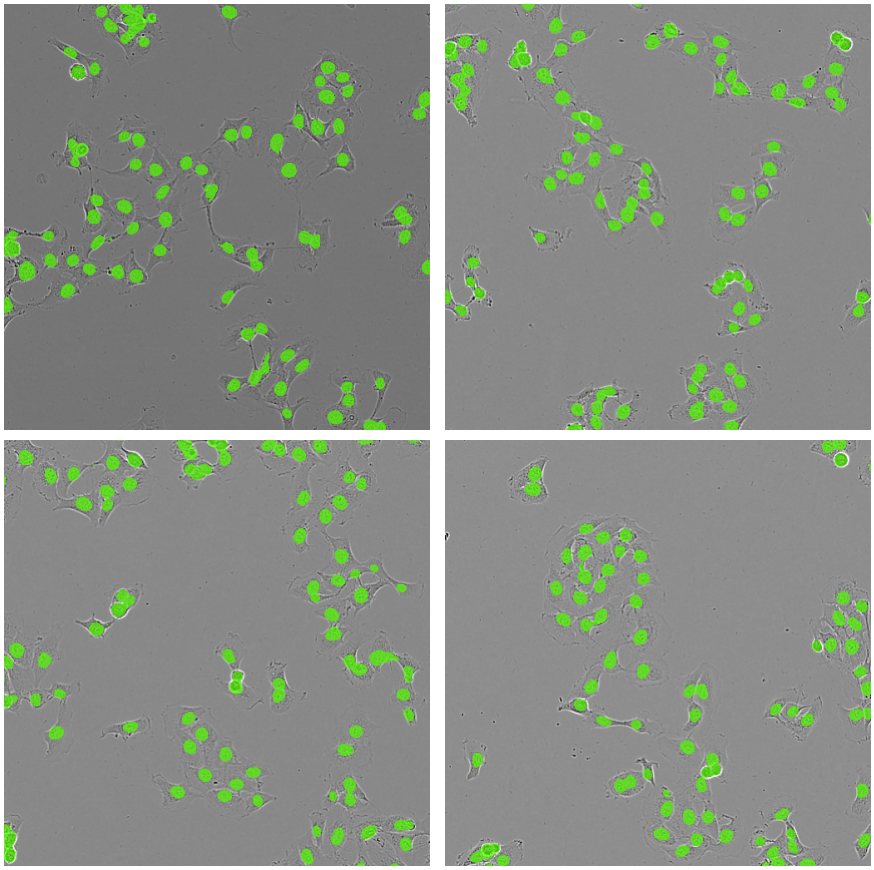

Figure A.1: A random sample of BJ-RAS phase-contrast images with prediction overlays.

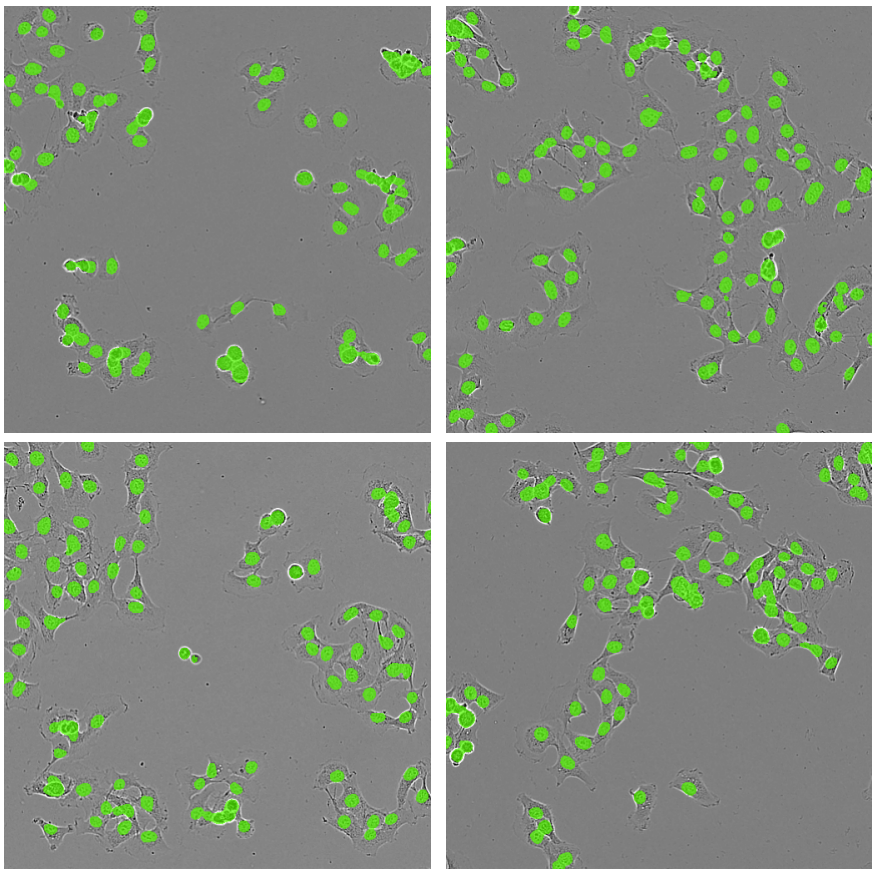

Figure A.2: A random sample of BJ-RAS phase-contrast images with prediction overlays.

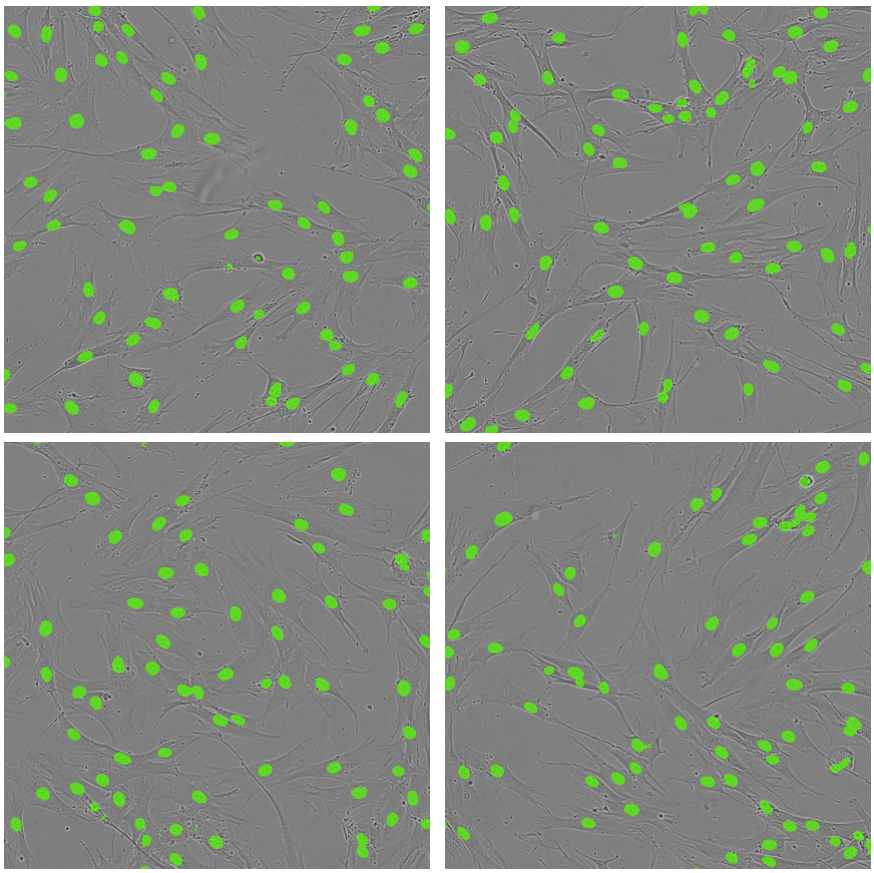

Figure A.3: A random sample of BJ-TERT phase-contrast images with prediction overlays.

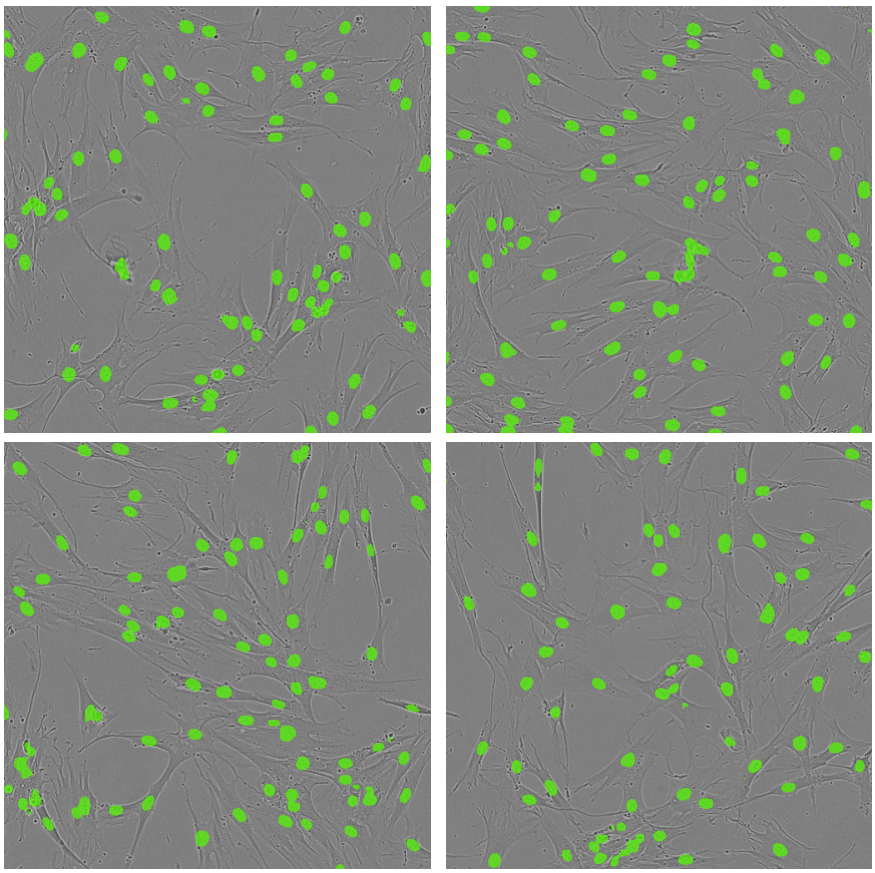

Figure A.4: A random sample of BJ-TERT phase-contrast images with prediction overlays.

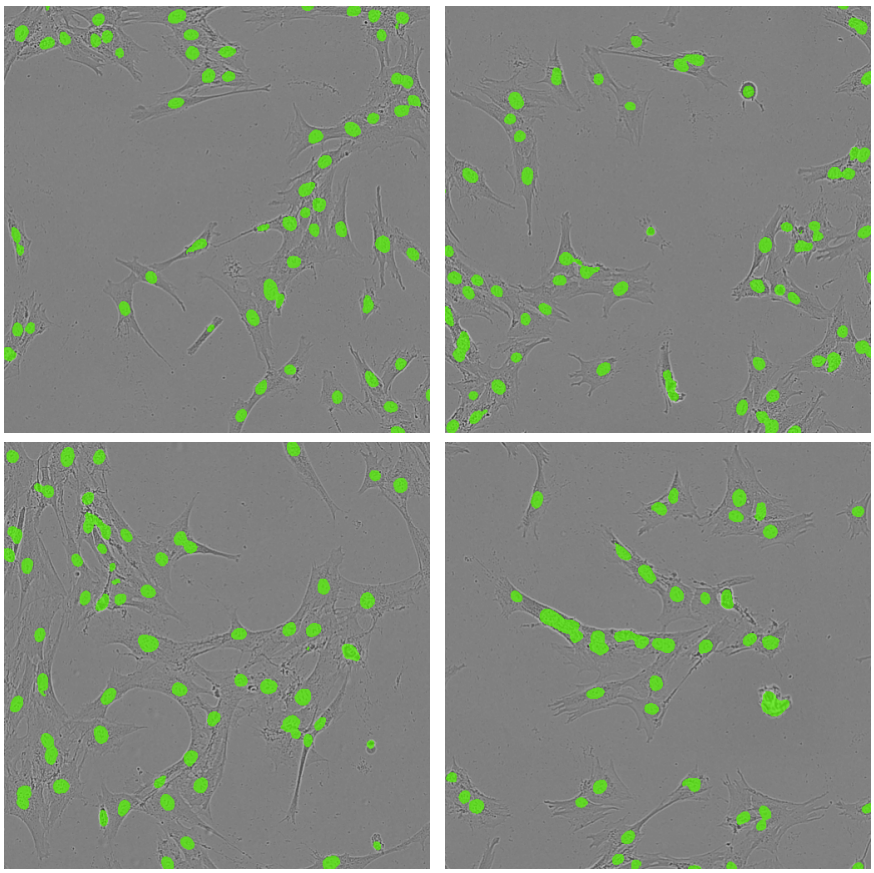

Figure A.5: A random sample of BJ-SV40 phase-contrast images with prediction overlays.

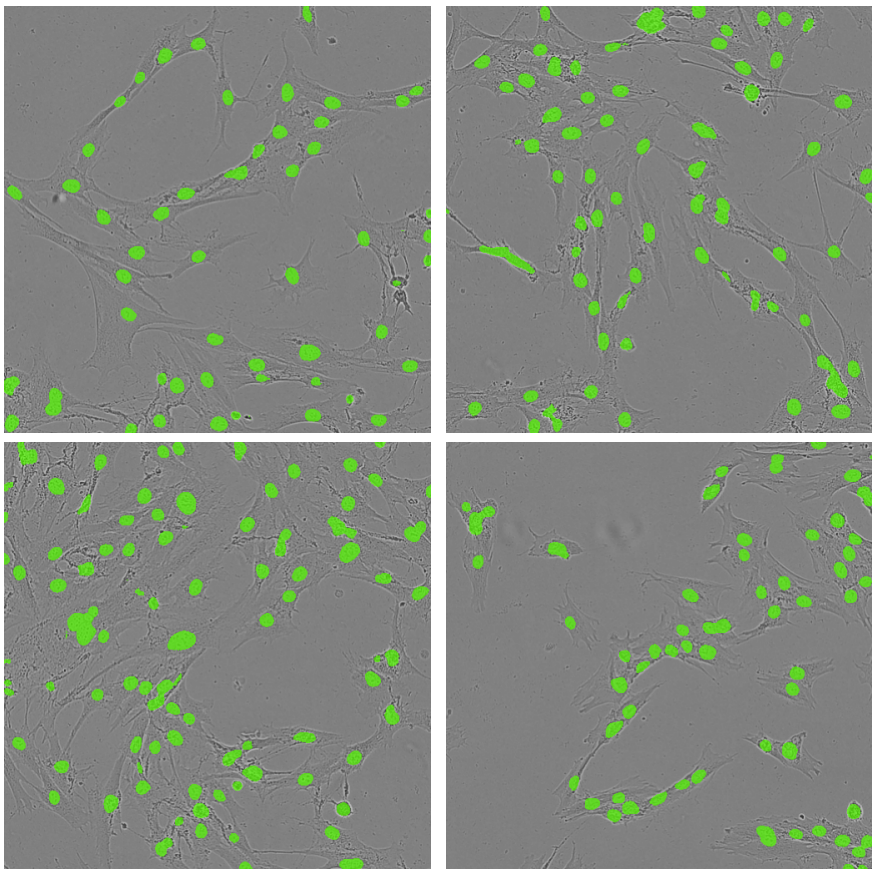

Figure A.6: A random sample of BJ-SV40 phase-contrast images with prediction overlays.

Printed by Tryckeriet i E-huset, Lund 2023

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2023-939
http://www.eit.lth.se