UCLouvain

École polytechnique de Louvain

Applying Deep Convolution Neural Network (CNN) on flow cytometry data for the prediction of a survival type clinical outcome

Author: **Badr-Ali MOUADEN**
Supervisors: **Michel VERLEYSEN, Laurent GATTO**
Readers: **Philippe HAUCHAMPS, Cyril DE BODT**
Academic year 2022–2023
Master [120] in Data Sciences Engineering

Abstract

"*Applying Deep Convolution Neural Network (CNN) on flow cytometry data for the prediction of a survival type clinical outcome*", by Badr-Ali MOUADEN. *Université Catholique de Louvain*, Master's degree in data science engineering.

In 2020, a group of researchers published an innovative paper that presented a new approach to the processing of cytometry data. Employing a Deep Convolutional Neural Network architecture, they demonstrated the potential of these models to not only surpass existing automated biomarker identification methods but also provide fresh insights in cytometry data analysis. Drawing inspiration from this successful initiative, this Master's thesis sets out to validate the application of Deep CNN models to predict survival outcomes using flow cytometry data. The dataset chosen for this endeavor comprises 383 HIV-positive patients, destined to progress to AIDS. Among them, 191 were designated for training and 192 for testing, as part of an automated data analysis challenge that yielded a random forest model as the winning solution.
In pursuit of this goal, a modified version of the CNN model is introduced, tailored to the unique aspects of survival data through an adapted loss function—the negative log-likelihood. The thesis delves into the design of the architecture and the intricate process of hyperparameter selection. Model performance evaluation is carried out using the concordance index, which measures prediction accuracy. During testing, the emergence of significant overfitting issues prompted the exploration of remedies, resulting in the incorporation of dropout layers and data augmentation techniques to mitigate these.
The pinnacle of the proposed approach materializes in a deep CNN model with two convolutional layers housing 6 feature maps each, followed by an average pooling layer and 3 fully-connected layers. This optimized model was evaluated on the test set and compared to the winning model's performance. While our model's performance falls slightly short of the winner's, it remains remarkably competitive. By embracing this thesis, we have substantiated the potential of deep CNNs within the realm of cytometry data analysis, underscoring the need for further exploration and refinement in this direction.

KEYWORDS : flow cytometry, survival analysis, deep convolutional neural network, concordance index , HIV/AIDS, negative-log likelihood

Résumé

"*Application de réseaux de neurones profonds (CNN) sur des données de cytométrie en flux dans le but de prédire le résultat d'un essai clinique de type survie*", par Badr-Ali MOUADEN.
Université Catholique de Louvain, Master en Sciences de l'ingénieur en data science.

En 2020, un groupe de chercheurs a publié un article innovant proposant une nouvelle approche pour le traitement des données de cytométrie. Utilisant une architecture de réseau neuronal convolutif profond (deep CNN), ils ont démontré le potentiel de ces modèles non seulement pour surpasser les méthodes existantes, mais aussi pour apporter de nouvelles perspectives dans l'analyse des données de cytométrie. S'inspirant de cette initiative réussie, ce mémoire vise à valider l'application de modèles de deep CNN pour prédire des résultats lié à une étude de longévité en utilisant des données de cytométrie en flux. Le jeu de données choisi pour ce mémoire comprend 383 patients séropositifs pour le VIH, destinés à évoluer vers le SIDA. Parmi eux, 191 ont été désignés pour l'entraînement et 192 pour le test, dans le cadre d'un challenge d'analyse de données automatisée ayant donné lieu à un modèle random forest en tant que solution gagnante.
Dans le cadre de ce mémoire, une version modifiée du modèle deep CNN est introduite, adaptée aux aspects uniques des données de survie grâce à une fonction de coût adaptée - la negative log-likelihood. Le mémoire aborde la conception de l'architecture et le processus complexe de sélection des hyperparamètres. L'évaluation des performances du modèle est effectuée à l'aide de l'indice de concordance, qui mesure la précision des prédictions. Lors des tests, l'émergence de problèmes de surajustement significatifs a incité à explorer des solutions, aboutissant à l'intégration de couches de désactivation (dropout) et de techniques d'augmentation de données pour atténuer ce problème.
Le modèle le plus abouti, un modèle de CNN profond avec deux couches de convolution abritant chacune 6 feature maps, suivies d'une couche de pooling moyenne et de 3 couches entièrement connectées, est proposée. Ce modèle est évalué sur le jeu de test et comparé aux performances du modèle gagnant. Bien que les performances de notre modèle soient légèrement inférieures à celles du modèle gagnant, elles restent remarquablement compétitives. En embrassant ce mémoire, nous avons validé le potentiel des CNN profonds dans le domaine de l'analyse des données de cytométrie, soulignant la nécessité de poursuivre l'exploration et le perfectionnement dans cette direction.

KEYWORDS : Cytométrie en flux, analyse de survie, réseau de neuronal convolutif profond, indice de concordance, VIH/SIDA, negative-log-likelihood

Acknowledgement

Within the following pages, you will discover the results and findings of this scientific endeavor—a collective achievement forged through collaboration and the support of many individuals. In this concise and non-technical section, I would like to extend my gratitude to those who have been important in this journey.

Foremost, I am grateful for the individual with whom I shared an intensive collaboration. This person provided invaluable assistance, offering solutions during moments of uncertainty and skillfully guiding me through each phase. My thesis work being a small part of his PhD project, he played a pivotal role in shaping the research topic. Navigating the complexities of this intricate dataset was no easy task, yet he generously dedicated time from his workweek to ensure my alignment with objectives and to offer invaluable advice. Our regular exchanges paved the evolution of this work. Thus thanks to M. Philippe Hauchamps, his guidance has been indispensable, and I owe much of my progress to his support.

I also wish to express profound appreciation to Professors Laurent Gatto and Michel Verleysen, supervisors of this thesis. Their support and expert guidance have been invaluable throughout this research journey. Despite their demanding schedules, they graciously made time to engage with me, sharing their insights and even challenging me to deliver bi-weekly presentations in preparation for the oral defense. I am particularly grateful to Prof. Gatto for granting me the privilege of working within his department.

A special acknowledgment goes to Prof. John Lee, whose invaluable contributions introduced fresh perspectives for approaching the research problem. Our discussion yielded profound insights, which find resonance in the concluding remarks, notably the concept of the histogram CNN, an idea uniquely attributed to him.
In addition, I extend my heartfelt gratitude to my girlfriend, Noémie, whose steadfast support and comforting words provided relief and perspective during challenging times. Her presence alleviated the journey's burdens, and her unwavering patience and encouragement during the final stretch were deeply cherished.

Lastly, I would like to thank my family and Noémie's family for their consistent words of encouragement and steadfast support, which provided an essential foundation throughout this journey.

Table of Contents

List of Figures

Introduction

In the realm of global health, few challenges have left as deep a mark as the Human Immunodeficiency Virus (HIV) and its devastating aftermath, Acquired Immunodeficiency Syndrome (AIDS). AIDS is a condition where the immune system weakens over time, making individuals susceptible to various infections and cancers. Despite significant progress in medical research, HIV/AIDS continues to pose a significant global health issue, highlighting the importance of finding innovative approaches to tackle its complexities.
HIV/AIDS has cast a long-lasting shadow, impacting millions of lives across the world. Recent estimates indicate that as of 2022, approximately 39 million people are affected by this disease [1]. While access to antiretroviral therapy has improved for many, about 24% of those affected still lack access [2], resulting in nearly 630,000 AIDS-related deaths in 2022 [1]. These numbers serve as poignant reminders of the ongoing significance of this disease.

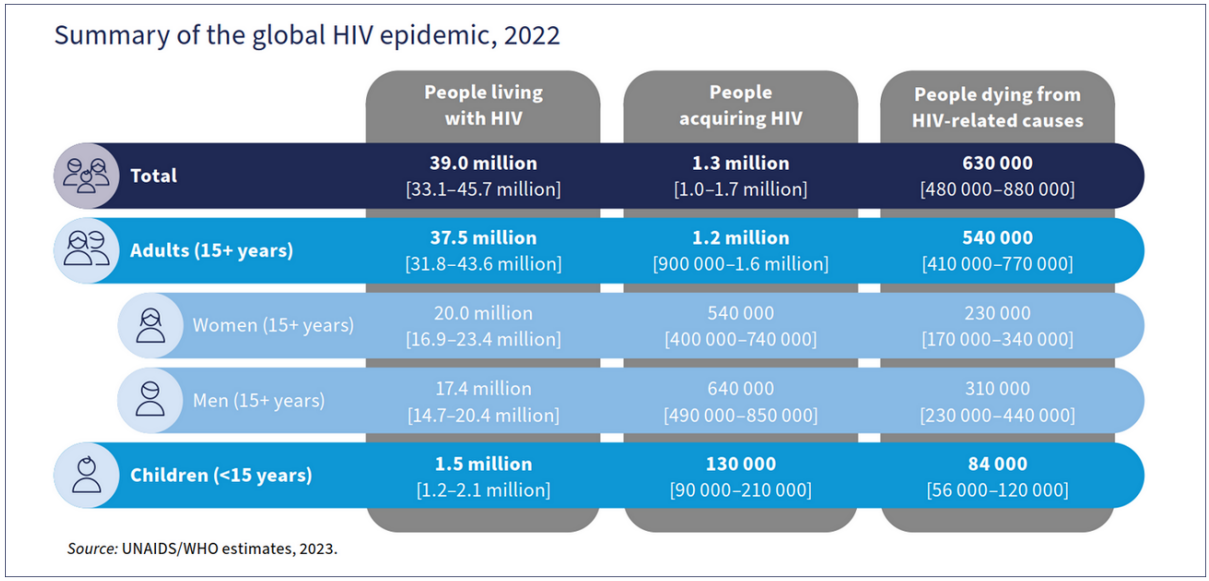


Figure 1: Some statistics about HIV/AIDS.
Reproduced from [1].

In this intricate landscape, where medical knowledge meets technological innovation, our thesis sets out on a journey that combines Deep Convolutional Neural Networks (CNNs), survival analysis, flow cytometry, and the intricate task of predicting how HIV-positive patients' health will evolve as they move closer to AIDS progression.

In the realm where medical insights converge with technological advancements, the FlowCAP consortium emerges, a collaborative effort aimed at bridging the gap between intricate flow cytometry data and tangible clinical outcomes. Flow cytometry, a state-of-the-art technique employing lasers, allows us to meticulously explore cell populations, offering a profound understanding of immune responses at the cellular level. This groundbreaking technology holds immense promise in unraveling diseases like HIV/AIDS. Within this sphere, the FlowCAP IV competition assumes a pivotal role. It challenged teams of researchers to construct models capable of predicting a significant clinical outcome: the time it takes for AIDS progression to occur [3]. The data originates from a study [4] on 383 individuals living with HIV for which flow cytometry data were collected.

This dataset brings together two ingredients: cellular markers and survival analysis. The latter, which falls under the umbrella of statistics, assists in understanding the durations involved and uncovers the hidden stories behind events as they gradually unfold over time. For us, survival analysis isn't just a method; it's a way to interpret the uncertainty nature of survival data. As we work on this puzzle, we face an important challenge: how can we predict the length of an event while dealing with incomplete information?

Navigating this intricate landscape, we encounter a new player, the deep CNN. Inspired by a success story in predictive analytics, where these networks successfully predicted binary outcomes (is the subject CMV [1] positive) [5] on cytometry data, we ask ourselves an exciting question: can Deep CNNs help us connect flow cytometry data to survival outcomes?

As we delve into these subjects, our thesis aims to push the boundaries of innovation. We hold two primary objectives: firstly, to adapt the architecture of the network that previously predicted binary outcomes [5], enabling it to predict survival outcomes; secondly, to implement and assess this adapted model using the HIV dataset [4]. These endeavors are aimed at demonstrating the potential of deep CNNs to surpass existing state-of-the-art models.

[1] Cytomegalovirus

1 Outline of this work

This thesis is structured as follow :

- Chapter 1 delves into the core of neural networks, explaining the components and methods from neurons to backpropagation. We explore layer types, activation functions, and loss functions. Convolutional Neural Networks (CNNs) are introduced, along with dropout layers and batch normalization.
- Chapter 2 explores survival analysis, covering censoring, survival functions, hazards, and cumulative hazards functions. The Kaplan-Meier estimator and Cox Proportional Hazards method are highlighted. Finally the concordance index is depicted as a good metric for survival analysis.
- Chapter 3 uncovers the dataset, focusing on flow cytometry data, stimulated/unstimulated data, and markers. Preprocessing steps and censored patient ratios are discussed, culminating in insights drawn from Kaplan-Meier curves.
- Chapter 4 reviews notable papers, including the CNN paper for binary outcomes on CyTOF data that inspired this thesis [5] and other CNN-based survival outcome studies on images.
- Chapter 5 details our experiments, from the validation protocols to the baseline model architecture and hyperparameters. We address overfitting, culminating in test set evaluation.
- Chapter 6 reflects on findings, limitations, and future improvements.

Part I

Background knowledge

Chapter 1

Neural Networks

1.1 What are neural networks ?

The content of this section is adapted from [6], [7] and [8].

Neural networks (NNs) are a class of machine learning models. They are inspired by the structure and functioning of the human brain and have proven to be highly effective in solving a wide range of complex problems such as classification, prediction or even art creation. They are trendy nowadays with the explosion of deep learning which is a subset of neural networks and has shown to be very valuable in a number of tasks. Their use is very widespread currently and is showing in everyone's today life such as in the very trendy ChatBot **ChatGPT**.

1.1.1 Neurons

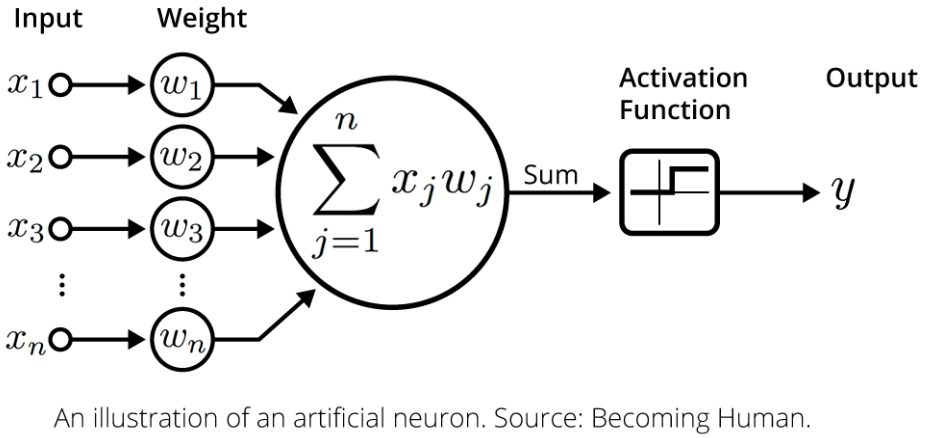


Figure 1.1: Functioning of a neuron.
Reproduced from [9].

Neural networks are composed of interconnected artificial neurons. A neuron receives one or multiple inputs and associate to each of them a weight which will indicate how important it is in order to solve the problem at hand. Inputs can be, depending on the

layer the neurons is on (which will be explained later), outputs from other neurons or coming from external data. It is important to notify that the inputs can only be real numbers to allow for a mathematical computation. This means that in order to input an information such as the color of a car, a preprocessing is compulsory in order to modify the color names to real numbers.
The neuron will then calculate its output based on this equation :

$$y = \varphi\left(w_0 + \sum_{j=1}^{n} w_j x_j\right) \tag{1.1}$$

where y denotes the output, the x_j are the inputs and w_j are the associated weights, w_0 is called the bias, it is a constant added and is also a parameter of the neuron, φ is the activation function.

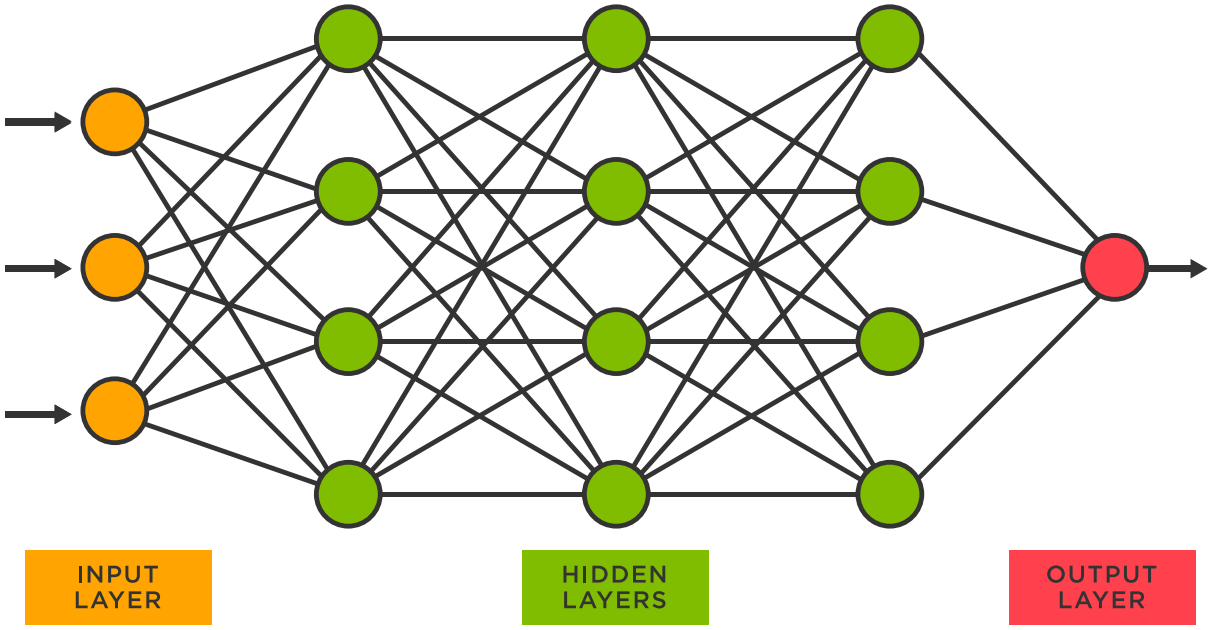


Figure 1.2: Example of an artificial neural network.
Reproduced from [10].

Neurons are arranged in 3 types of layer :

- **Input layer** : these neurons receive the raw data, which can be numerical, categorical, or even image data. Each neuron in the input layer represents a specific feature or attribute of the data (after suitable preprocessing).
- **Hidden Layers** : these layers are contained between the input and output layer. They are where the bulk of the neural network's computation takes place. Each neuron in a hidden layer receives inputs from either the input layer or another hidden layer and passes its output to another hidden layer or the output layer.
- **Output layer** : these neurons are responsible to output the final prediction or result of the neural network. Their output is what will be used by the programmer to solve its task. The number of neurons in the output layer depends on the

nature of the problem; for example, a binary classification task would have one output neuron which would output -1 for one class and 1 for the other, while a multi-class classification task would have multiple output neurons.

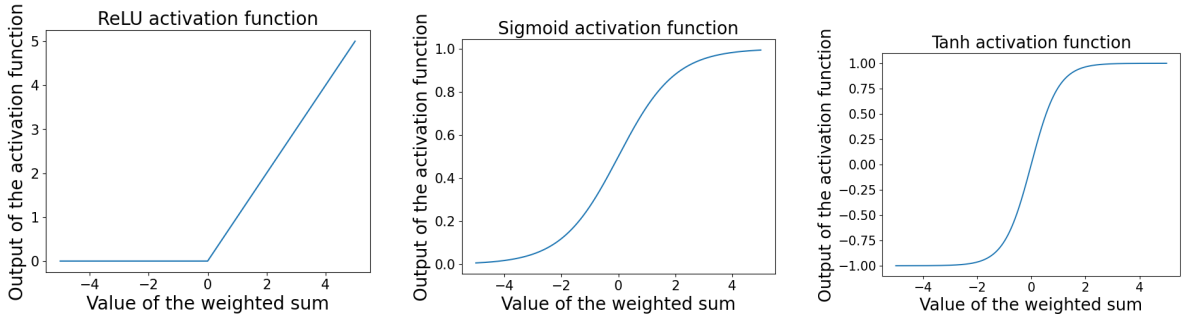
1.1.2 Activation function

The activation function is a crucial component of artificial neural networks. It is a function that is calculated on the weighted sum inside a neuron. Most neural networks use non-linear activation functions but linear ones exist too. Linear functions are not that used because if all neurons used linear activation functions, the entire network would essentially behave like a single linear transformation. This is because the combination of linear functions remains linear. As a result, the network would not be able to learn complex patterns and non-linear relationships in the data, severely limiting its expressive power. Despite that, some layers of a network could use a linear activation function.
Activation functions can range from a step function to more complex functions such as sigmoid.
Here are some of the mostly used activation functions :

- **ReLU** (Rectified Linear Unit): defined as $\varphi(x) = \max(0, x)$, it sets all negative values to zero and leaves positive values unchanged. ReLU biggest strength is that it is computationally efficient and helps alleviate the problem of vanishing gradient (which will be explained later).

- **Sigmoid** : defined as $\varphi(x) = \frac{1}{1+e^{-x}}$. It compresses the input into a range between 0 and 1, which makes it useful for binary classification problems.

- **Tanh** (Hyperbolic Tangent) : defined as $\varphi(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$. It compresses the input into a range between -1 and 1 which is also a good choice for classification.



(a) ReLU function. (b) Sigmoid function. (c) Tanh function.

Figure 1.3: Graphs of most used activations functions.

1.2 How do we train neural networks ?

The content of this section is adapted from [6], [7], [8] and [11].

In order for a neural network to learn how to solve a task, it has to be trained, so that it makes a prediction based on these data. Once the predictions are outputted, the neural network can then learn by receiving a score based on how it has performed in the task. Based on this score, the neural network will then change the value of its weights across all the layers. Two key components of training are thus in place, the so-called loss function (or cost function) which will be responsible for the scoring and the backpropagation process which is used to feed back to every neurons the changes to be made on its weights. One phase of training which is called an **epoch** is thus a succession of predictions made by the neural network, evaluation by the loss function and backpropagation through the network.

1.2.1 Loss function

A loss function is a function used to measure the dissimilarity or error between the predicted outputs of a model and the actual ground truth labels. The primary objective of the model during training is to minimize/maximize this loss function, as doing so indicates that the model's predictions are becoming more accurate.
Most loss function will compute a value based on the predictions of the model and the true labels. For example, a well-known loss function for prediction tasks is the RMSE (root mean square error) which will compute the square root of the average distance between the predictions and the true labels. A smaller value will indicate that the model behaves better [12].

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}} \tag{1.2}$$

where $\hat{y}_i$ denotes a prediction and y the true label associated for a sample, n is the size the dataset.
It is important to note that depending on the task at hand, a particular loss function will be better that others and the choice of the latter is a fundamental part to achieve good performance. For example, binary cross entropy is more useful for binary classification tasks. It measures the difference between predicted probabilities and actual binary labels. Mathematically, the binary cross entropy loss is calculated as [13]:

$$\text{Binary Cross Entropy Loss} = -\frac{1}{n}\left[\sum_{i=1}^{n} y_i \log(p(y_i)) + (1 - y_i)\log(1 - p(y_i))\right] \tag{1.3}$$

where $p(y_i)$ is a predicted probability for a binary event, y_i, the corresponding true binary label and n is the size of the dataset.

Another important aspect is that the goal is to minimize/maximize this function and, to achieve that goal, a variant of the gradient descent method is used. By forcing the

model in the direction of the strongest descent/ascent of the loss function, one can expect that a future iteration of the model will get a lower/higher score of the loss function making it better. This process is done until the gradient of the loss function at the value that the model has scored, is null (ideally) or close to zero. This means that a good loss function must be differentiable across its domain limiting the amount of possibilities.

1.2.2 Backpropagation

Backpropagation is a process used to train neural networks. It plays a crucial role in adjusting the network's weights to minimize the difference between the predicted outputs and the actual target outputs during the training process. The term "backpropagation" stands for "backward propagation of errors," as the algorithm calculates and distributes the errors from the output layer back to the network hidden layers and the input layer. The name stems from the fact that a model can only be scored (by a loss function) based on its output prediction and not on the output of any preceding layers. This means that to feed back the information to the previous layers, an algorithm is needed to tell each neurons to which amount of error it is responsible for. Backpropagation is an efficient way to compute the gradients of the loss function with respect to each weight in the network. These gradients represent the direction and magnitude of the adjustments needed to minimize/maximize the loss.

The typical update formula for a certain weight vector $\mathbf{w}$ at a certain epoch t is :

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla J(\mathbf{w}_t) \tag{1.4}$$

where α is known as the learning rate and $\nabla J(w_t)$ is the gradient of the loss function J with respect to the weight $\mathbf{w}$ at epoch t.
To efficiently compute these gradients associated with each weight, the chain rule is used, propagating the errors backward from the output layer to the hidden and the input layers, adjusting the weights at each step.

1.3 Convolutional neural network (CNN)

The content of this section is adapted from [14].

1.3.1 Motivation

CNNs are a specialized type of artificial neural network designed primarily for processing and analyzing visual data, such as images and videos. They have become the go-to architecture for computer vision tasks due to their ability to effectively capture spatial patterns and hierarchical features in images.
Traditional neural networks, while effective for many tasks, have limitations in handling images directly. Images have a high-dimensional structure (pixels arranged in rows and columns), and standard neural networks require a large number of parameters to process them effectively. This leads to increased computation time and a risk

of overfitting (memorizing the training data rather than learning general patterns). CNNs address these issues by leveraging the concept of convolution.

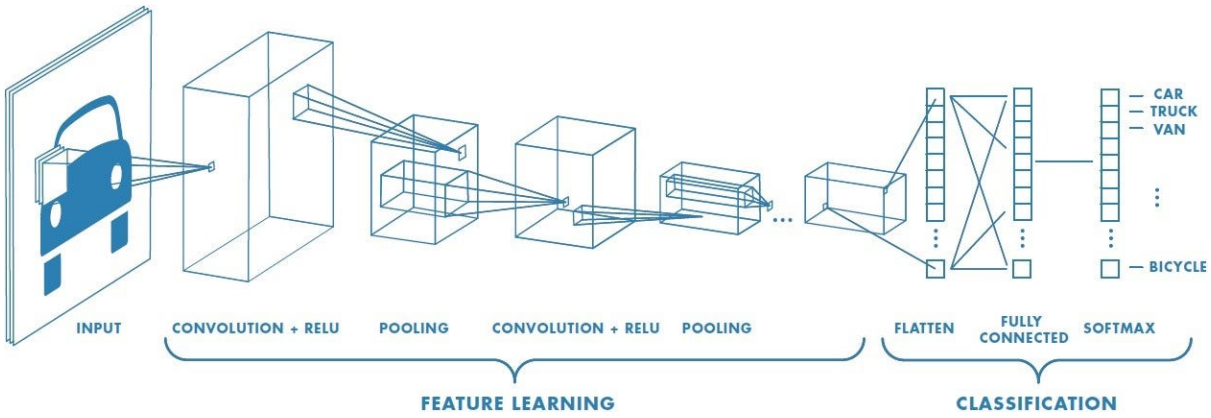


Figure 1.4: Example of a CNN.
Reproduced from [15].

1.3.2 Convolutional layers

The key component of a CNN is the convolutional layer. Instead of having fully connected neurons, these layers use small filters (also known as kernels or feature detectors) that slide over the input image. Each filter captures a specific pattern or feature, such as edges, corners, or textures.

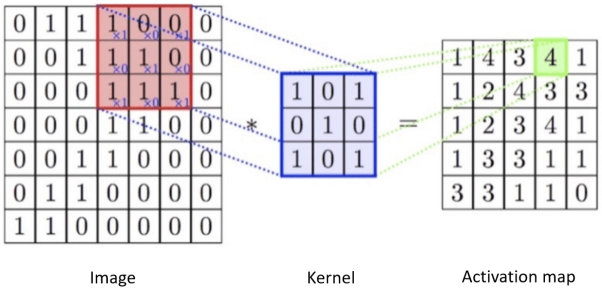


Figure 1.5: Example of a convolution in a convolutional layer.
Reproduced from [16].

During the convolution operation, the filter multiplies its weights with a local region of the input image and sums them up before passing them through an activation function. The filter moves through the whole input image producing an activation map as shown in figure 1.5. The result represents the presence of particular features in different locations of the input. The size of the kernel and the amount of movement the kernel does when it slides are 2 parameters of the convolutional layer called respectively the kernel size and stride.

1.3.3 Pooling layers

After convolution, CNNs typically use pooling layers to reduce the spatial dimensions of the activation maps while retaining essential information.
There is typically two type of pooling operations, the max pooling or average pooling. The first one will, across a window of defined size and stride, select the maximal value while the latter will compute the average. This reduces the computational complexity and helps the network focus on the most relevant features.

1.3.4 Fully connected layers

After several convolutional and pooling layers, CNNs often incorporate fully connected layers to make the final predictions as shown in figure 1.4. First the output of the final pooling layer will be flattened in order to get a one dimensional tensor then fully connected layers will be added. These layers connect all the neurons from the previous layer to the current layer, eventually leading to the output layer, where the final predictions are made. Those layers reassembles the more classical ones described in subsection 1.1.1.

1.3.5 Dropout layers

The fully-connected layer, as previously mentioned, establishes connections with all weights in the preceding layer, which can be a substantial number. Consequently, this architecture can exhibit overfitting tendencies, hindering the network ability to generalize effectively to new data. To address this, various techniques are employed to mitigate overfitting, with one prevalent approach in Convolutional Neural Networks (CNNs) being the integration of dropout layers, pioneered by Hinton [17]. As implied by the term, the dropout layer induces the network to "drop" certain nodes during each iteration, determined by a designated probability known as the keep probability, typically ranging from 0.2 to 0.5 [18]. This keep probability signifies the likelihood of a specific node being omitted during training. During backpropagation, the weights linked to these dropped nodes remain unchanged. These nodes are then reintroduced for the subsequent iteration before another set is selected for dropout, promoting robustness and aiding in countering overfitting.

1.3.6 Batch normalisation layers

The content of this section is adapted from [19].
In deep neural networks, the distribution of inputs to each layer can change as training progresses. This can lead to the "internal covariate shift," where the distribution of inputs to a layer keeps shifting, making it difficult for subsequent layers to learn effectively. Batch normalisation is used in that context to improve training stability and convergence by normalizing the activations of intermediate layers. It was introduced as a way to mitigate issues like the slow convergence but also the vanishing gradient [20] problem which is a challenge in deep learning where gradients become extremely small during backpropagation. This causes earlier layers to update slowly,

impeding the learning process and leading to slower convergence or stagnation [21]. These problems often arise in deep networks

Given a batch of inputs: $\{x_1, x_2, ..., x_m\}$, where m is the batch size, for a specific layer's activation, the batch normalisation process involves the following steps:

1. Compute the mean (μ) and variance (σ^2) of the activations across the batch.

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{1.5}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2 \tag{1.6}$$

2. Normalize the activations within the batch using the mean and variance.

$$\hat{x_i} = \frac{(x_i - \mu)}{\sqrt{\sigma^2 + \epsilon}} \tag{1.7}$$

   where ϵ is a small constant added to avoid division by zero.

3. Introduce learnable parameters γ (scale) and β (shift) to scale and shift the normalized activations, allowing the network to learn appropriate scaling and shifting.

$$y_i = \gamma\hat{x_i} + \beta \tag{1.8}$$

4. During training, the parameters γ and β are learned through backpropagation, just like other network parameters.

1.3.7 Advantages of CNNs

The CNNs utilize the same filter weights across different regions of the input, reducing the number of parameters and promoting feature reuse. They also use filters that slide over the input and are able to detect features regardless of their location in the image, making them translation invariant. And finally, CNNs learn hierarchical representations of visual data, where early layers detect simple features and later layers combine them to identify complex patterns.

Chapter 2

Survival data and analysis

The content of this chapter is adapted from [22], [23].

2.1 Introduction

In medical research, the analysis of time-to-event data is of utmost importance to understand the progression of diseases, assess treatment outcomes, and predict patient survival. During medical studies, it is not rare to have patients that do not experience the event, in those cases survival analysis is a important set of methods to get an understanding of the matter at hand.
In the survival domain, an *event* is defined as the occurrence of the outcome under study. For example, it could be the remission from a cancer or the death of a patient. The time to the event is typically called the *survival time*.

2.2 Censoring

Most methods will not be effective in the clinical study environments. A common and inevitable issue when dealing with time-to-event data is that the event of interest could occur beyond the study's observation period or is unknown for some subjects. This is called **censoring** and those subjects are called censored. Censoring is a fundamental challenge in survival analysis, and understanding how to handle it is essential for accurate and meaningful analysis. In the latter part of this chapter, methods that handle those types of data will be presented.
There are two main types of censoring [24]:

- **Left-censoring** : it occurs when the event of interest is known to have occurred before a specific time point (start of the study for example). However, the exact event time is unknown. This type of censoring is relatively less common but may arise in situations where the study began after some events had already occurred. Left-censored observations provide information only about the maximum time to the event for those subjects.
- **Right-censoring** : it occurs when the event of interest has not occurred for some individuals by the end of the study or observation period. In other words, these

individuals are still "alive" or "event-free" at the end of the study. It could also be an individual that dropped out of the study before it has ended without experiencing the event so far. Right-censoring is the most common type of censoring. Right-censored observations provide information only about the minimum time to the event for those subjects.

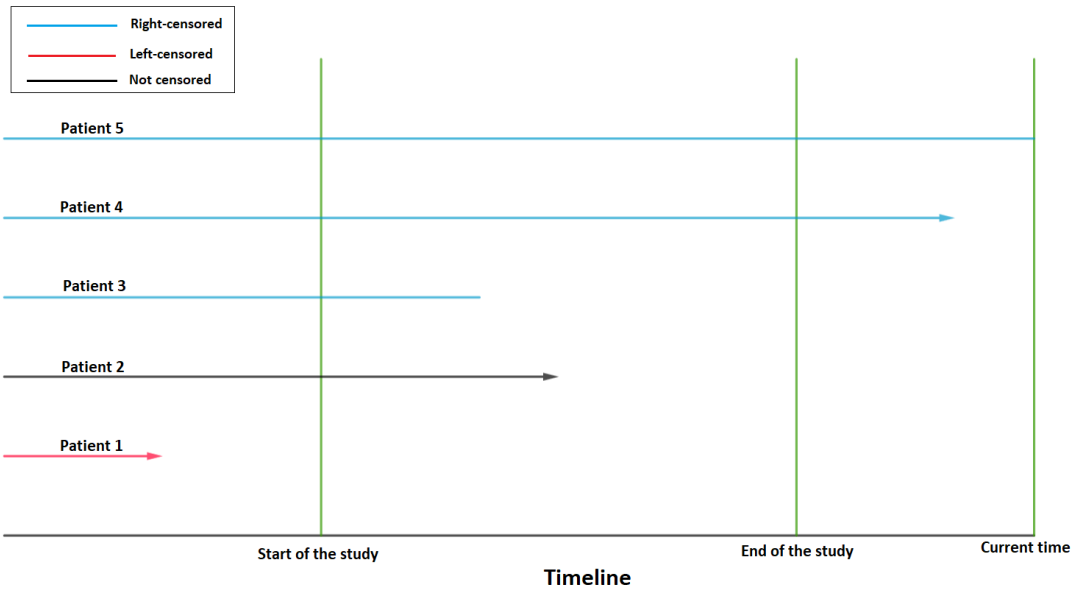


Figure 2.1: Example of left and right censoring.
Adapted from [24].

2.3 Mathematical formulation

In order to handle such type of data, models have been developed and will be presented in latter sections. First of all, a description of the most common functions is important to lay the basics for the models.

2.3.1 Survival function

The survival function (defined as $S(t)$) is a fundamental concept in survival analysis. It represents the probability that an individual will survive beyond a specific time point t. Mathematically, the survival function is defined as:

$$S(t) = \mathrm{P}(T > t) \tag{2.1}$$

where T is the time to the event. It describes the probability that an event will occur after time t. Obviously, the survival function at time $t = 0$ is $S(0) = 1$ indicating that

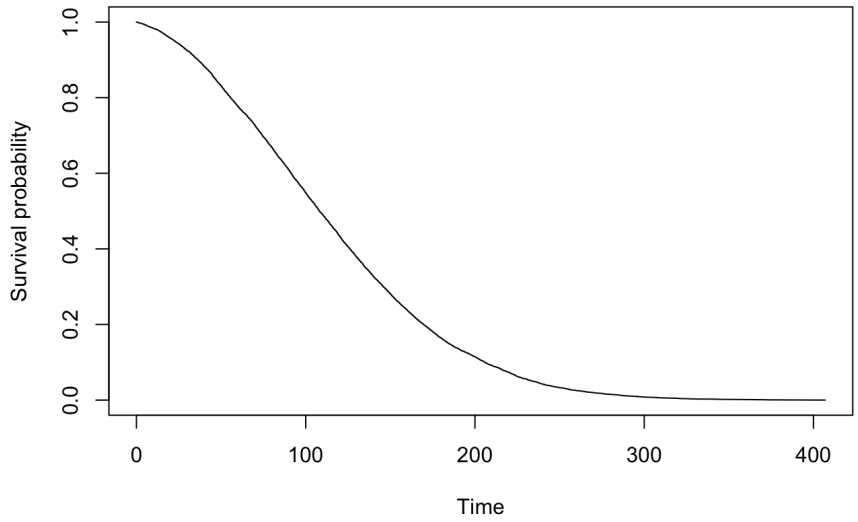


Figure 2.2: Example of a survival function.
Reproduced from [22].

the probability of surviving beyond time zero (time zero is arbitrarily set in time) is 1. As time progresses, the survival probability gradually decreases. With the same logic, this function is null at $t = \infty$. A possible survival function is shown in figure 2.2.

2.3.2 Hazard function

The hazard function (defined as $h(t)$) is another important function in survival analysis. Mathematically, it is defined as:

$$h(t) = \lim_{\Delta t \to 0} \frac{\mathrm{P}\left(t \leq T \leq t + \Delta t \mid T \geq t\right)}{\Delta t} \tag{2.2}$$

where the numerator is the probability that the event occurs between t and $t + \Delta t$ given that the individual has survived up to time t. The hazard function is a measure of the risk or rate of occurrence of the event at each specific time point. It provides insights into how the risk changes over time. If the hazard function is constant over time, it indicates that the risk of experiencing the event is constant. However, in most real-world scenarios, the hazard function varies with time.

2.3.3 Cumulative hazard function

There is no easy way to quantify the hazard function. To bypass this problem, a widely used function to estimate the latter is the cumulative hazard function. We will show that it is closely related to the hazard and the survival function.
Defined as $H(t)$, it represents the accumulated hazard or risk of experiencing the event of interest up to time t. Mathematically, it is defined as the integral of the hazard

function $h(t)$ from time 0 to t:

$$H(t) = \int_0^t h(\tau)\, d\tau \tag{2.3}$$

It can also be linked to the survival function through the following relation relation :

$$S(t) = \exp\left(-H(t)\right) \tag{2.4}$$

2.4 Kaplan-Meier estimator and curve

The content of this section is adapted from [25], [26], [27].

The functions presented above gives us more insight on survival data but in order to use them, models are needed to estimate them.
The Kaplan-Meier estimator is a non-parametric method used to estimate the survival function $S(t)$ in the presence of censored data. It calculates the probability of surviving beyond each observed time point based on the available data, making it particularly suitable for datasets with right-censored observations. It accounts for decreasing number of individuals at risk as time progresses, ensuring a more accurate estimation of survival probabilities over time compared to assuming a constant number of individuals at risk.
This estimator is defined as :

$$\hat{S}(t) = \prod_{i:\, t_i \leq t} \left[1 - \frac{d_i}{n_i}\right] \tag{2.5}$$

where t_i is a time point where at least one event has happened, d_i is the number of events at time t_i and n_i is the number of subjects that have not yet experienced the event or have been censored at time t_i.
This estimator makes a certain number of assumptions which are important to point out :

- No subject have experienced the event at time $t = 0$, thus $\hat{S}(0) = 1$.
- Censored subjects have the same survival probability as those who are still in the study.
- Same survival probability for a subject that entered the study at the beginning or later.
- Times of recorded events are the real event times, it does not consider late recording.

The Kaplan-Meier estimator's results are often depicted using a Kaplan-Meier curve, which visually illustrates the behavior of the estimator over time. The x-axis denotes time, while the y-axis displays survival probabilities. The curve typically exhibits a stepwise pattern, reflecting changes in survival probabilities at observed event times.

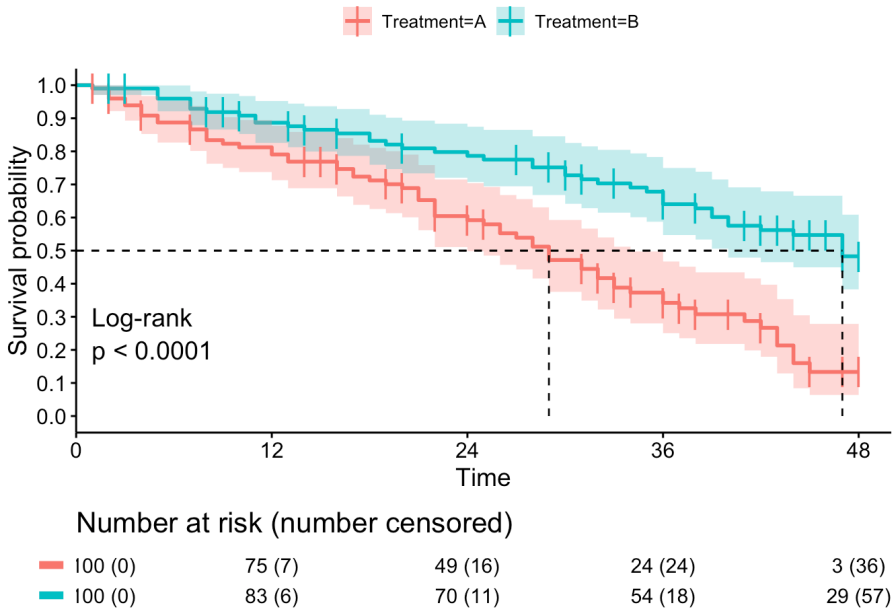


Figure 2.3: Example of a Kaplan-Meier curve.
Reproduced from [28].

In figure 2.3, an additional feature is introduced: short vertical segments are integrated into the curve, corresponding to the time points where subjects were censored. These lines offer a visual indication of censored data instances. This comprehensive graphical representation serves to enhance comprehension of the dataset as a whole, providing insight into both observed events and censored data points, contributing to a more comprehensive understanding of the survival data. For example, see the spread of censored subjects, are they more at the beginning of the study, the end or more spread out ? This will be an important aspect of this Master's thesis. Furthermore, the curve often incorporates confidence intervals (shades of blue and red around the plots on figure 2.3), which convey the uncertainty associated with the survival estimates, offering a range within which the true survival function is likely to lie. This provides a valuable measure of the precision of the Kaplan-Meier estimator results.

2.5 Cox proportional hazards model

The content of this section is adapted from [29], [30].

Another method, which is key for this work, is the Cox Proportional Hazards model (CPH). This model is a semi-parametric model that leverages information about the subjects, referred to as covariates, such as age, medical diagnoses, treatment types, genetic markers, and other relevant factors, to estimate the hazard function.

2.5.1 Mathematical expression

If we consider a set of p covariates (covariates vector) for subject i, $\mathbf{x_i} = [x_{i1}\ x_{i2}\ ...\ x_{ip}]^\top$, and $\boldsymbol{\beta} = [\beta_1\ \beta_2\ ...\ \beta_p]^\top$, a parameter vector, then Cox's model states that :

$$h_i(t) = h_0(t)\ \exp\left(\beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip}\right) = h_0(t)\ \exp(\mathbf{x}_i^\top \boldsymbol{\beta}) \tag{2.6}$$

where $h_0(t)$ is called the *baseline hazard function* and is the hazard function for a subject with a null covariates vector.
As one can notice, a strong assumption is made by the model, the ratio of the hazard functions between two different subject is independent of time $\frac{h_i(t)}{h_j(t)} = \frac{\exp(\mathbf{x}_i^\top \boldsymbol{\beta})}{\exp(\mathbf{x}_j^\top \boldsymbol{\beta})}$, this is why the model is called "proportional". This means that for every subject, the form of their hazard function is the same, the only difference is a scaling factor depending on their covariate values.
CPH is considered as a semi-parametric model because it leaves the baseline hazard function unspecified and estimates the model based on the partial-likelihood maximization. The derivation of this partial likelihood will not be detailed here but can be found in [31].
The partial likelihood is :

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{r} \frac{\exp(\mathbf{x}_i^\top \boldsymbol{\beta})}{\sum_{j \in \mathcal{R}_i} \exp(\mathbf{x}_j^\top \boldsymbol{\beta})} \tag{2.7}$$

where r is the number of events and $\mathcal{R}_i$ is the riskset of subject i which are all subjects that are still event-free and not censored at t_i which is the time to event of subject i.
It is equivalent and easier to maximize the *partial log-likelihood* $\ell(\boldsymbol{\beta})$.

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{r} \mathbf{x}_i^\top \boldsymbol{\beta} - \log\left[\sum_{j \in \mathcal{R}_i} \exp(\mathbf{x}_j^\top \boldsymbol{\beta})\right] \tag{2.8}$$

Equation 2.8 is quite central in this Master's thesis, it will be shown latter that it can be used as a loss function for a neural network with the covariate vectors as input and $\boldsymbol{\beta}$ being the neural network's weights.

2.6 Concordance index (C-index)

The content of this section is adapted from [32].

In the two previous sections, two methods typically used in survival analysis to estimate important functions that will inform domain specialists about survival outcomes were presented. But in order to test if a model is better than another one for a specific problem, one needs a metric.
The C-index quantifies the consistency between predicted and observed survival times, providing valuable insights into the model's ability to discriminate between individuals with different survival outcomes. It is a measure of the probability that predicted

survival times of two randomly selected subjects have the same ordering as their observed survival times $\mathrm{P}(\hat{t}_i > \hat{t}_j \mid t_i > t_j)$ with $\hat{t}_i$ and $\hat{t}_j$, the predicted survival times and t_i and t_j, the observed survival times.
The concordance index can be split in two parts, one that will quantify the relative ordering of event-event pairs and the other for event-censored pairs. It is important to note that it is not possible to measure that probability with two censored subjects because of the lack of conclusion that we can draw based on their censor time. Assessing the concordance of a event-censored pair is possible because if the censored subject has a censor time that is greater than the non-censored subject then a comparison is possible. Indeed we know for sure that until its censor time, the censored subject did not experience the event. However the opposite is not possible, if the censoring time is lower than the event time, there is no way to know for sure if the censored subject experienced the vent before, after or at the same time as the non-censored subject. Those pairs are thus not considered for the computation of the concardance index.

2.6.1 Mathematical formulation

Let's consider the random variables $o_{ij} = \hat{t}_i > \hat{t}_j \mid t_i > t_j$ that is equal to 1 if the pair ij is ordered (concordant pair) and 0 otherwise and k_{ij} which takes the value ee for an event-event pair and ec otherwise. Then one can show that [32]:

$$\frac{1}{\mathrm{CI}} = \alpha \frac{1}{\mathrm{CI}_{ee}} + (1-\alpha)\frac{1}{\mathrm{CI}_{ec}} \tag{2.9}$$

with

$$\mathrm{CI}_{ee} \equiv \mathrm{P}(o|ee) \tag{2.10}$$
$$\mathrm{CI}_{ec} \equiv \mathrm{P}(o|ec) \tag{2.11}$$
$$\alpha \equiv \mathrm{P}(ee|o) = 1 - \mathrm{P}(ec|o) \tag{2.12}$$

with $\mathrm{P}(o) = \mathrm{P}(o_{ij} = 1)$, $\mathrm{P}(ee) = \mathrm{P}(k_{ij} = ee)$ and $\mathrm{P}(ec) = \mathrm{P}(k_{ij} = ec)$ for simpler notation. CI_{ee} represents the part of the C-index for event-event pairs and CI_{ec} for event-censored pairs.
If we consider N^+, the number of concordant pairs, N^-, the number of discordant pairs and $N^=$, the number of ties then the C-index can be estimated from :

$$\widehat{\mathrm{CI}} = \frac{N^+ + \frac{1}{2}N^=}{N^+ + N^- + N^=} \tag{2.13}$$

where one of the most common way to handle ties in the predicted times is used, it considers that has half the chance to be correctly ordered and half not to be hence the $\frac{1}{2}$ in the formula.

2.6.2 Significance in model evaluation and comparison

The C-index is a very powerful metric to use in case of survival data because of their specific nature. It can assess a model's discriminatory power, i.e., its ability to differentiate between individuals with different survival outcomes. A higher value indicates

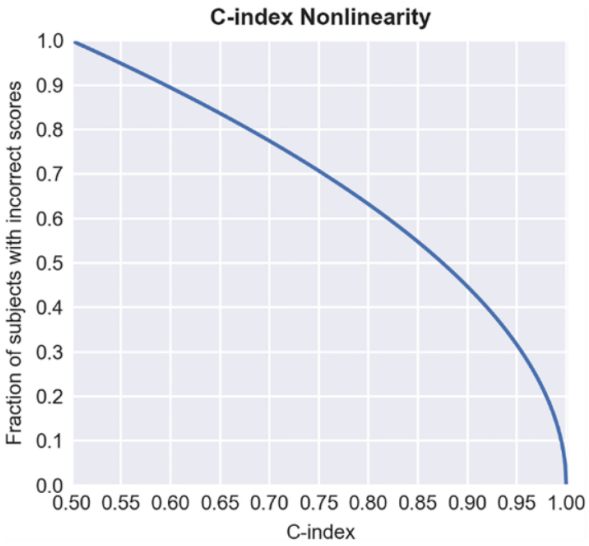


Figure 2.4: Relation between the concordance index and the ratio of subjects with incorrect scores.
Reproduce from [33].

better discrimination and more accurate predictions. It is intuitive and varies between 0 (completely discordant model predictions) and 1 (perfectly concordant model predictions), 0.5 represents a random prediction model. Because of that researchers can use it to compare multiple survival models and select the one with the highest discriminatory power (highest C-index). It aids in identifying the most suitable model for a specific dataset. One important thing to notice is that most of the time, in survival analysis, the accurate prediction of survival times or risk scores in itself is not that important. Most of the time, researcher will use survival models to discriminate between two drugs, find a link between a clinical outcome and a patient covariates and so on, thus making the exact prediction not that relevant but the ability to compare is primordial and well measured through the C-index.
Despite that, it is important to note that the relation between the concordance index and the number of subjects whose risk has been correctly predicted is not linear [33]. This means that as shown on figure 2.4 that to perfectly score half of the subject, a concordance index of more than 0.85 is needed.
This means that a 10% increase in concordance index is not always linked with a 10% increase in the number of correctly assigned scores.

Part II

Main

Chapter 3

Dataset

3.1 Introduction

The dataset under study in this Master thesis comes from a challenge that was organised by the FlowCAP (Flow Cytometry: Critical Assessment of Population Identifications Methods) consortium. The primary goal of FlowCAP is to provide a platform through competitions for researchers and bioinformaticians to assess and compare different computational algorithms and tools that are designed for various tasks within flow cytometry data analysis. These tasks may include cell population identification, clustering, data preprocessing, and visualization. Authors of that contest challenged teams around the world to study data coming from a clinical study that followed HIV+ subjects and wanted them to identify covariates that correlate the most to the clinical outcome (the progression to AIDS) [3]. These data are thus survival data. The other particularity about these data is that the covariates are high dimensional flow cytometry data.

3.2 What are cytometry data ?

The content of this section is adapted from [34], [35]

Cytometry is a field at the intersection of biology, physics, and engineering. It encompasses a wide range of techniques used to analyze and quantify individual cells or particles within complex populations. These techniques allow researchers to obtain quantitative data, providing valuable insights into cellular characteristics and behavior. Cytometry has played a pivotal role in advancing our understanding of diverse biological processes, including immunology, hematology, oncology, microbiology, and drug discovery. Cytometry is mostly used on blood cells samples taken from a patient under study to extract information such as size, count, complexity, phenotype, viability and functionality. Cytometry is a single-cell analysis which means that characteristics are measured in each cell separately.

3.2.1 Flow cytometry

Flow cytometry is a subset of cytometry in which cells or particles suspended in a fluid stream pass through an illuminated detection area one at a time, a process known as hydrodynamic focusing. Laser-based light sources provide excitation energy, which interacts with the cells, resulting in scattered and fluorescent emissions. The emitted light is then collected and measured by detectors, generating electrical signals that are converted into digital form for further analysis.

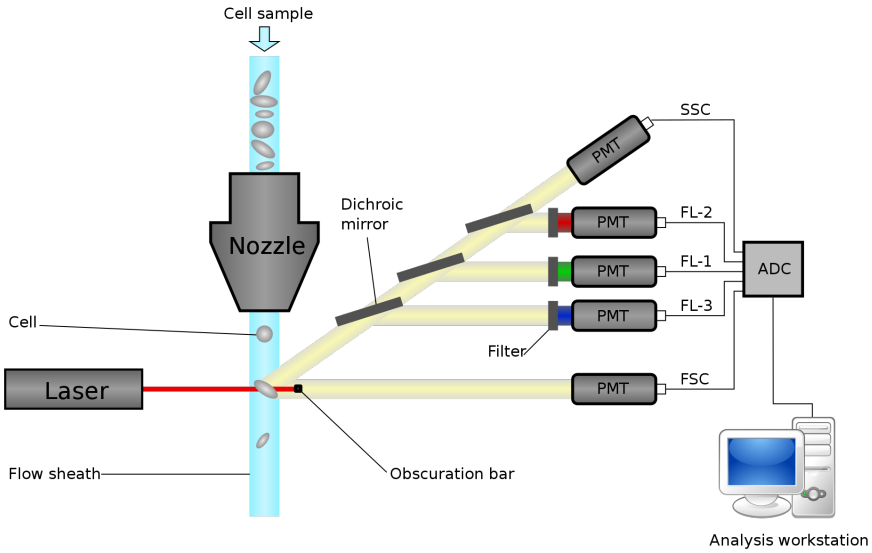


Figure 3.1: Flow cytometry basic components.
Reproduced from [36].

Multiple light signals are measured. The 2 principals are the FSC and SSC which are respectively the *forward scatter* and *side scatter*. The first one is a measure related to the size of the cell while the latter is related to the content [37]. Then a series of other emissions are collected, they are produced by fluorescent markers. These are fluorescent antibodies that binds with specific surface proteins of interest to the researchers that will react to the laser light. To analyze these markers, medical professionals collect a volume of blood for each subject. Thus the number of cells analysed through flow cytometry is varying for subject to subject. As shown on figure 3.2, we thus obtain for each individual subject, measurements of a certain number of markers measured p for each cell of their blood sample. Flow cytometry dataset are thus a sort of table for each patient with p markers on the columns (which are the same for each subject) and n cells on the rows (which varies from patient to patient). Also the cell populations present in each blood sample could be quite different and one cannot know, before examining, what the proportion is and whether this proportion is different from patient to patient.

Flow cytometry data					
	Marker 1	Marker 2	Marker 3	...	Marker p
Cell 1					
Cell 2					
Cell 3					
...					
Cell n					

Figure 3.2: Structure of flow cytometry data for one patient.

3.3 Structure of the data

The study [4] was made on 383 subjects but only 191 were made available for the challengers and thus it was decided that models created for this thesis could only be trained on the 191 subjects. This dataset comes in multiple files. To begin with, there is one file (a meta data file) that contains the information about the subjects :

- the **subject index**, number identifier between 0 and 382
- the **event indicator**, it takes a value 1 if the subject is not-censored, 0 otherwise,
- the "**time**", this value is counted in days and is the the *survival time* if the subject is not-censored and the *censor time* otherwise,
- the **name of the subject data files**, there are two files, one with stimulated cells and one without stimulation (see section 3.6)

Some patients are credited with a negative time, this is because their samples were taken shortly after the diagnosis of progression to AIDS. Over the 383 subjects, 9 are in this situation with 3 in the 191 subjects that were given for training. Each subject is associated with two files containing cytometry data, one with stimulated cells and one without stimulation (this will be discussed in section 3.6). Each subject file is composed of a table as shown in figure 3.4 with over the columns, the ***markers*** and some other information, i.e., the time of acquisition of the data and the cell ID in the last 2 columns which are not important and each line represents a ***cell*** of that subject. There is in total, 16 markers that were measured and that are available to use.

Because those data are cytometry data, the number of cells for each subject is variable and the order of those cells is not important because the cell type is not known a priori, and therefore cannot be used as a grouping/classifying variable. After a preprocessing (which will be developed in section 3.5), the subject file with the smallest amount of

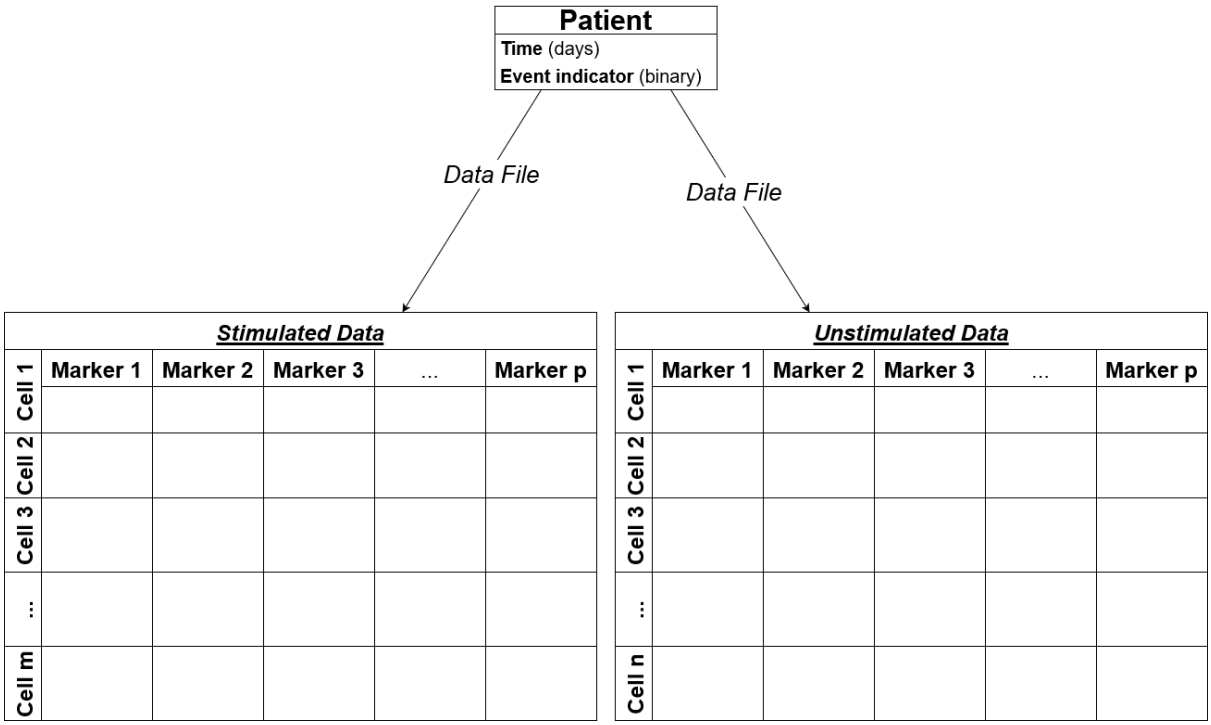


Figure 3.3: Structure of a patient's data. It is considered that there is **m** cells in the stimulated file and **n** in the unstimulated one, both have **p** markers.

	FSC-A	FSC-H	SSC-A	IFNG	TNFA	CD4	CD27	CD107A	CD154	CD3	CCR7	IL2	CD8	CD57	CD45RO	VIVID/CD14
0	0.911234	0.709715	1.922913	0.787777	0.886715	2.497021	1.299668	1.111943	1.064137	1.799209	0.798372	1.266359	1.279310	1.183526	2.925874	2.678634
1	0.751652	0.621246	1.963637	1.025284	1.147213	1.120909	0.667109	0.926986	1.123812	0.820540	0.816798	0.950994	0.832061	2.009721	2.072834	0.765739
2	1.762417	1.419261	1.518023	1.145546	1.038522	0.758933	1.002476	1.025027	0.952050	2.321634	0.705537	1.191796	1.588646	0.957139	2.297120	1.330824
3	0.815535	0.678191	1.867652	1.808318	0.595204	2.516541	0.732788	1.966367	0.788216	1.989789	0.693127	1.168968	1.634078	0.920709	2.886754	2.591686
4	1.486742	1.131460	2.310294	1.077339	1.008299	1.279907	0.749898	0.970413	1.165376	2.458590	0.394045	1.400526	1.602960	0.923517	2.677228	1.736953
5	1.674084	1.199250	1.756959	1.257691	0.776808	2.839291	2.358543	0.812007	1.465921	2.741673	0.778039	1.623906	0.869037	0.954590	1.304191	0.857052
6	1.430942	1.157762	1.775792	0.931690	0.809820	2.602194	1.814789	1.133517	1.292527	2.758576	0.706358	1.779513	0.386644	0.883023	1.239118	1.128896
7	0.965837	0.780660	2.147994	0.843279	1.426063	1.307887	0.888479	0.930378	1.013453	2.132617	1.024698	0.943657	1.607565	1.233471	2.769516	2.065856
8	0.920368	0.625331	2.107427	1.079047	1.549305	3.047489	0.983918	1.328570	1.516981	2.502730	1.305006	0.990413	0.372807	1.127547	3.178905	1.272614
9	1.085919	0.890802	1.849370	1.095045	0.558415	2.996691	1.855477	1.104504	1.524865	2.741310	1.246017	1.515672	0.536758	0.605748	1.814404	0.772244

Figure 3.4: Sample of a subject data file.

cells contains 11821 cells, while the one with the largest amount contains 864540 cells. This is a very big difference which plays an important role in this work.
It is also important to note that each file is between 3 and 250 Mo. If we consider that we need to load in memory 2 times 191 data files, the total needed storage is about 27 Go. This is not possible for most computers because most GPUs cannot manage models that treat this amount of data. A workaround is needed and will be explained in a bit more detail in the chapter 5.

3.4 A bit more explanation about markers

As stated in the previous chapter, there are 16 markers which have all a role in discriminating between different cell population.

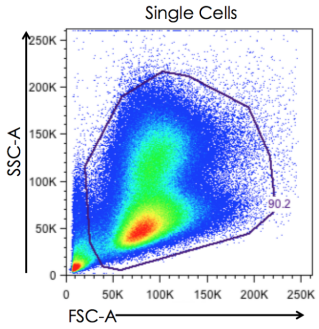


Figure 3.5: Flow cytometry plot. Reproduced from [38].

Figure 3.5 shows the typical technique that bioinformaticians use to explore the flow cytometry data, called *gating*. They will select some markers and plot them, some clusters will form and they then can isolate a cell population of interest as can be shown in figure 3.5 with FSC and SSC as discriminating markers. They reiterate the process with other markers to find sub-populations and so on.

Markers can be used as indicators of the size of the cells or the alive status of a cell for example. Some are present on only a specific type of cells while others are present on all but with variation in their expression. Some are on/off while others have a range of possible values. All of this complexity is present in this work data. As all data that are measured, some measurement errors can occur and to avoid feeding these improper data to a model, a preprocessing was used.

3.5 Preprocessing

A preprocessing was done on the cytometry data. The purpose was to eliminate non-useful information such as measurements error and make some transformations and compensations. This section will not go deep in the preprocessing because it was not the purpose of this work and was done by another person. A brief explanation will be made to ensure that the reader is aware that manipulations were done on the dataset. The steps of pre-processing are as shown on figure 3.6 :

- **Remove margins** : cells that have values of markers that are outliers, very high or low values are removed. A hypercube is created and everything that is not inside is removed.

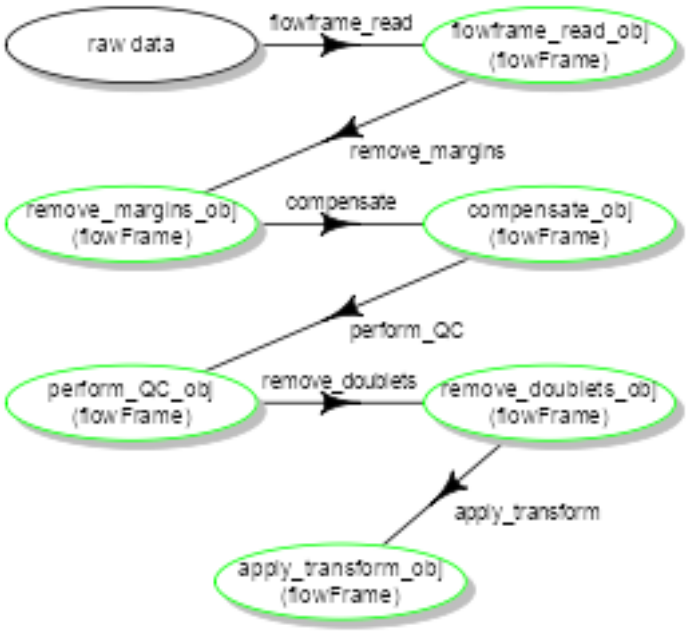


Figure 3.6: Steps of preprocessing. Provided by M. Hauchamps.

- **Compensation** : a linear transformation is performed on the fluorescent marker values, in order to compensate for the spilling over of fluorescence light from one channel to the other channels.
- **Remove QC** : it stabilises the signal through time, eliminating parts that are not stable using a tool named **PeacoQC** [39].
- **Remove doublets** : use the FSC channels (A and H, for area and height) to keep only the single cells.
- **Scale transformation** : use a biexponential function, i.e. a transformation that is similar to log transformation but can handle negative values [40], to compress the data to have a more discriminatory distribution as shown in figure 3.7.

A last step, consisting of automatically eliminating the dead cells, was also applied at first, but turned out to be non robust. Indeed, in some cases, this preprocessing cuts down the number the number of cells to 2-3 for some patients which is clearly not usable to make predictions. Therefore, it was decided to remove this pre-processing step.

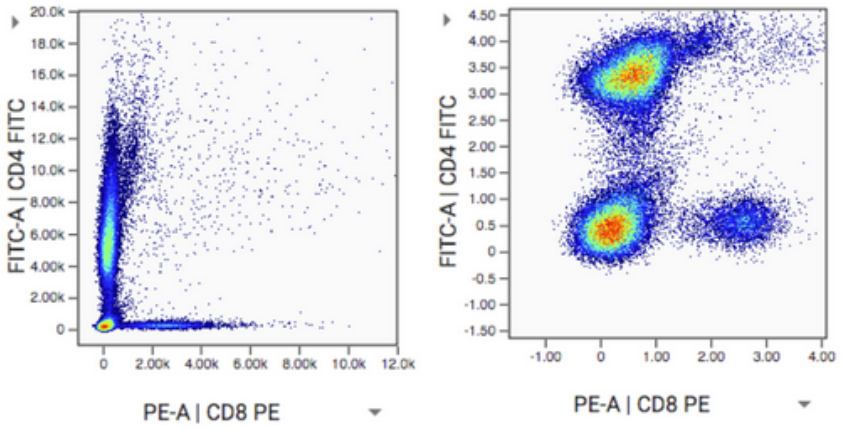


Figure 3.7: Left : no scale transformation. Right : scale transformation applied. Reproduced from [41].

3.6 In vitro stimulated and non stimulated samples.

As mentioned before, the cytometry data comes in two files per patient. One file contains intensities for cells that are stimulated and one without stimulation. The file without stimulation is just a blood sample that was taken from a patient and that was passed though a flow cytometer. The other is also a blood sample, but this one was stimulated in vitro with HIV antigens. This is made to create a response from specific cells in the context of HIV and can help identify the progression to AIDS. In association with unstimulated cells, researchers can identify those specific cells and can get a better understanding of the disease. Both stimulated and unstimulated cells is most probably related to the outcome and will thus be used in conjunction in this work.

3.7 Censoring

The dataset originates from a study about the time of progression to AIDS [4], it is a survival dataset. As stated in chapter 2, one key element is the presence of censored patients. They don't hold the same amount of information as the uncensored ones and thus need to be known for and analysed.

Of the 191 patients available for this thesis, only 34 are uncensored patients, which is only 17.8% of the dataset. This proportion is quite low. Censored patients give partial information and to have so many could worry us in the ability to find a relation between covariates and the outcome. Indeed, in order to predict the risk of a patient to progress to AIDS at any given point, we need informations about patients that lived beyond that time point as shown for example by the CPH model in section 2.5. The problem arises when a lot of patients are censored at the beginning of the study because they give us access to information until they leave the study. We need to make sure that from those censored patients, a good portion followed the study as long as possible so that later events can be predicted. To check that we can use the Kaplan-

Meier survival curve and see where the event happen and when patients are censored.

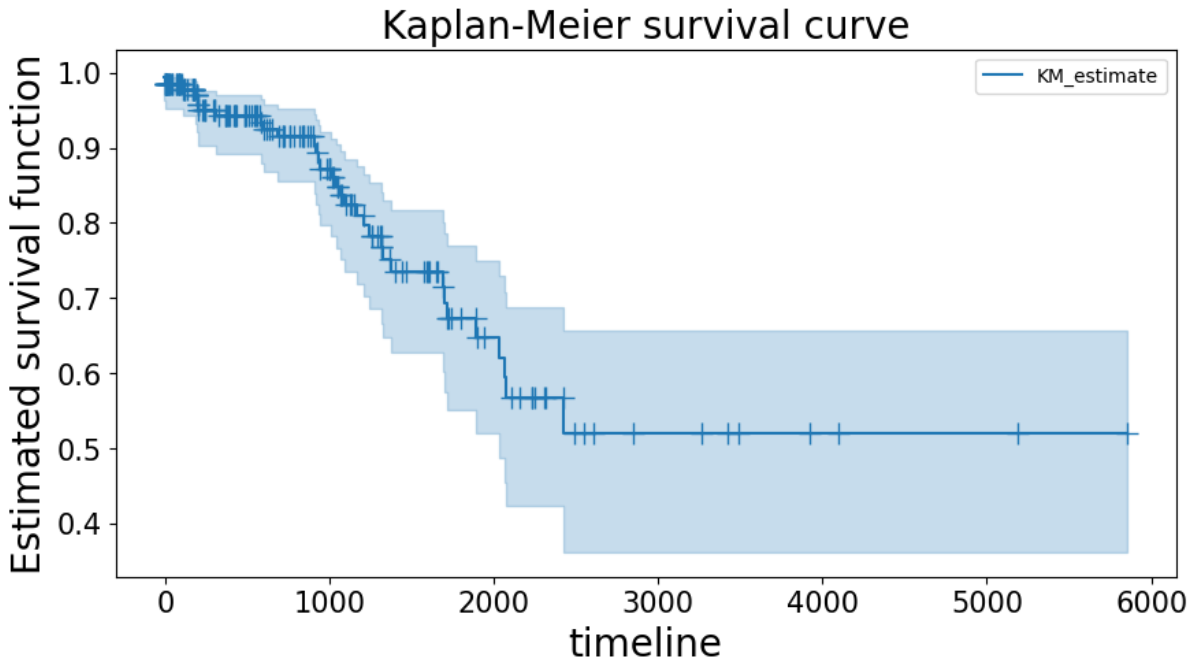


Figure 3.8: Kaplan-Meier survival curve of the dataset.

Days	0	1000	2000	3000	4000	5000	6000
At risk	188	75	24	7	3	2	0
Censored	0	100	137	150	154	155	157
Events	3	16	30	34	34	34	34

Table 3.1: Number of subject in each category at certain time steps

We can notice multiple things on this graph. First of all, table 3.1 shows at certain timesteps (every 1000 days) the number of subjects that had the event (*Events*), the number of subjects that are censored (*Censored*) and the number of subject that are still event free and not censored (*At risk*). Then, the vertical segments on the plot correspond to the time points where subjects were censored showing their distribution through time. We can see that at time $t = 0$, already 3 patients have already had the event (this was stated and explained earlier in section 3.3). We can also see that all the events happen before 3000 days and only 7 patients are left in the *At risk* group, they all are censored by the end of the study. Because there is no event past about 2500 days, the survival probability stays quite high for all time $t > 2500$ at about 0.5.

3.8 Conclusion

From the examination of the dataset, performed in the previous section, we can draw the following conclusions. First of all, the high amount of censored patients will be a challenge by the fact that they hold less information than the events. On the good side, a lot of the censored subjects have a late censoring time (20 are still in the study after 2000 days while 30 out of the 34 events already happened at that time as shown on table 3.1). Another challenge will be to deal with very high amounts of data (as suggested in section 3.3) and varying sizes of cells between patients whilst not knowing in advance the type of cell (cell population) that we are working with. The answers to these challenges will be discussed in chapter 5.

Chapter 4

CNNs used in cytometry and survival data

4.1 Introduction

In this chapter, some articles that explored the subject of artificial neural network associated with either cytometry data or survival data are reviewed. For all of the presented models, an open source implementation in Python can be found. The first one is implemented by the authors of the article and will be the basis of this thesis.
This chapter will be separated between a CNN model that was developed for cytometry data and in a second part, 3 articles will be presented with models that tackle the challenge of using CNNs in the case of survival data.

4.2 Deep CNN model for cytometry data

In this paper [5], the authors are working on mass cytometry data (another single cell cytometry technique that can access more markers called CyTOF) with 40 markers. The goal of the implemented model was to predict the CMV (cytomegalovirus) diagnostic status of patients coming from 9 different studies. This is a problem of binary classification (either the patient has CMV or not). The cytometry data has the same form as the data used in this work, it is for every patient a matrix with cells as rows and markers as columns.
They propose a CNN model with 2 convolutional layers. The size of the first convolutional layer is $1 \times n_{\text{markers}}$ so it will compute a value across all markers for each cell with 3 different sets of weights and thus create 3 features maps of size $n_{\text{cells}} \times 1$. This is because as stated in the chapter 3, there is no link from line to line in cytometry data and thus it is not logical to have a filter size that will be across a certain number of cells. The use that they make of the CNN is different than its traditional use in imagery. The kernel goes through the markers dimension because they are ordered in a certain way and this will not change from patient to patient so features that can be extracted this way are the same for all patients. The second convolutional layer works quite similarly, it will take row by row the values of the 3 feature maps created by the last layer and combine them creating at the end 3 new feature maps of size $n_{\text{cells}} \times 1$.

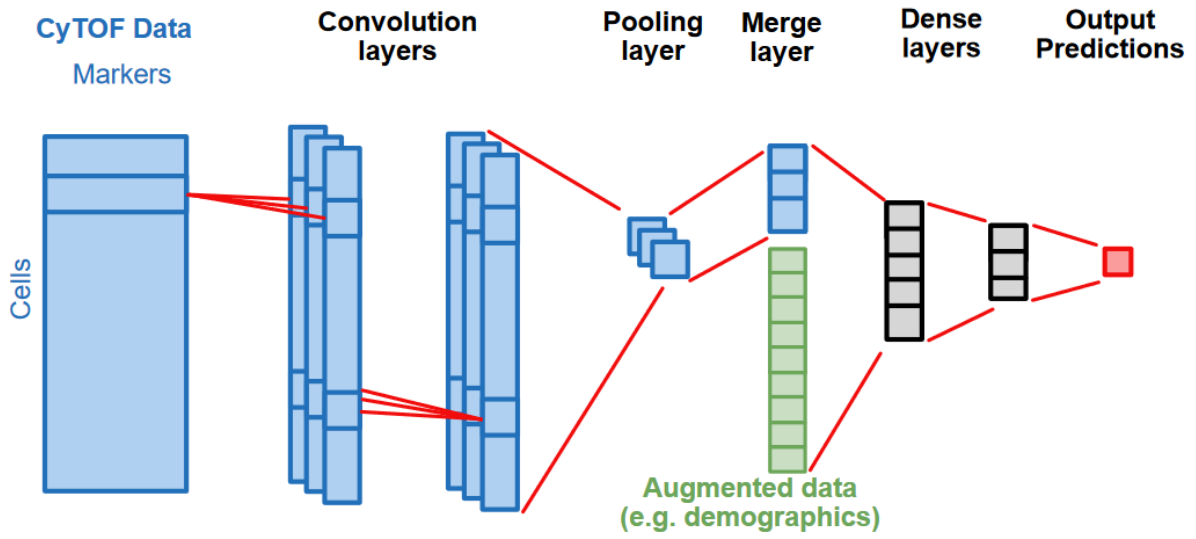


Figure 4.1: Structure of the CyTOF model
Reproduce from [5]

The goal of the convolutional layers is to combine multiple markers together to create features that are related to the outcome. Indeed, bioinformaticians use the cytometry data to isolate cell population that are correlated to the outcome by gating. This is a way to reproduce this logic. The activation function of those layers is **ReLU**.
After that a mean or max pooling layer is applied to transform the cell features into patient's features, they then add some more traditional data (age, sex,...) and finish with 3 layers of fully connected neurons decreasing in size to finish with the output layer containing 1 neurons that will output the class. They also used batch normalisation layers after each layer apart from the input and the pooling layer (see subsection 1.3.6). They used **ReLU** (see subsection 1.1.2) for all dense layers except the last one that is **sigmoid** to be able to output a binary prediction. The loss function used is a traditional one when it comes to predict a binary outcome, i.e. the binary cross-entropy (presented in subsection 1.2.1).

4.3 CNNs used on survival data

The first model is very interesting for this work because it works with cytometry data but the problem is that the outcome is quite easy to predict for neural network (classification is a task that is very widespread). The following models that will be presented try to tackle the challenge of predicting survival outcomes with CNNs. Despite that, they are more traditional in the data that they use, as all of them are CNN based on images, which is also widespread.

The models presented in this section are based on the CPH model and will all be predicting a time-to-event which is traditional in survival analysis. In order for their models to predict a hazards function, they all use as loss function a modified version

of the CPH partial log-likelihood, i.e. the negative partial log-likelihood [42], [43], [44]:

$$l(w) = -\sum_{i=1}^{r} h_w(\mathbf{x}_i) + \log\left[\sum_{j \in \mathcal{R}_i} \exp(h_w(\mathbf{x}_j))\right] \tag{4.1}$$

where $h_w(\mathbf{x}_i)$ is the output of the network which is called *risk scores*. These are a quantitative measures that reflect an individual's overall risk of experiencing the event over time. w are the model weights and $\mathbf{x}_i$, the input covariates. This function (in Equation 4.1) is to be minimised in order to optimise the network.

DeepConvSurv [42] is a CNN based model used on a lung cancer dataset of 450 patients with images and clinical information. It uses a CNN with 3 convolutional layers, 2 max pooling layers (with filter sizes that are more traditional because they work with images) and 1 fully connected layer, the loss function was modified from traditional ones to the negative partial log-likelihood described in Equation 4.1. To test out their model, they used 5 fold cross validation (explained in subsection 5.1.4) which is a know method for assessing the performance of a model [45].

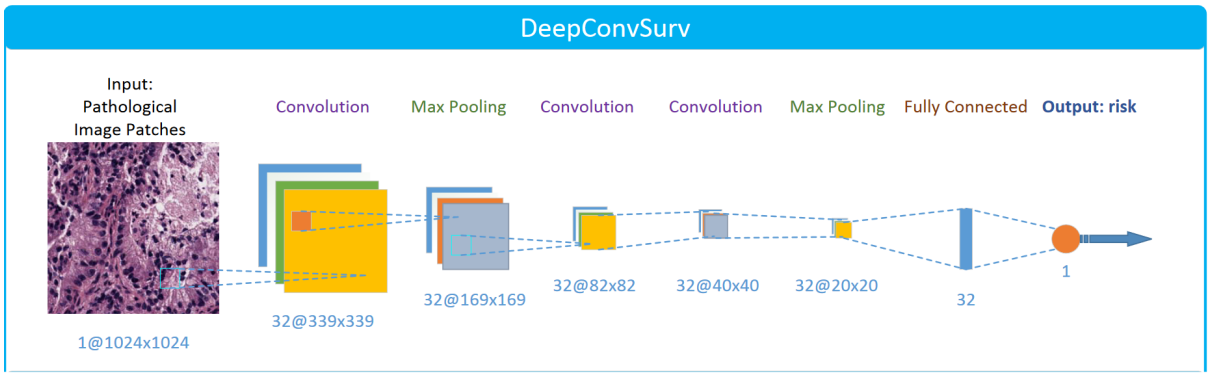


Figure 4.2: DeepConvSurv architecture.
Reproduced from [42]

It resulted in better results than a CPH model using, as covariates, information drawn from the images.

The next model [43] is also a model trained on images (tumor images). Researchers use 2 datasets. The first one is a simulated dataset generated based on MNIST dataset [46] of 10000 images where they randomly selected half to be non-censored as validation of their model on images. The second one is a rectal cancer dataset of 84 subjects which is used to validate their model on real data. It also uses Equation 4.1 as a loss function to optimise the model. It incorporates dropouts layers which are know to be a useful tool to avoid overfitting [47] (see subsection 1.3.5). The model contains 3 convolutional layers followed by a specific pooling layer called spatial pyramid pooling layer which is made to accommodate to different tumor sizes, then there is 2 fully connected layer and a output layer. They also used data augmentation on the rectal dataset which is a method known to help with overfitting and helping to have better performance out of the model [48]. Authors of this model used respectively a 10 and

5 fold cross validation protocol for the simulated and the rectal cancer dataset.

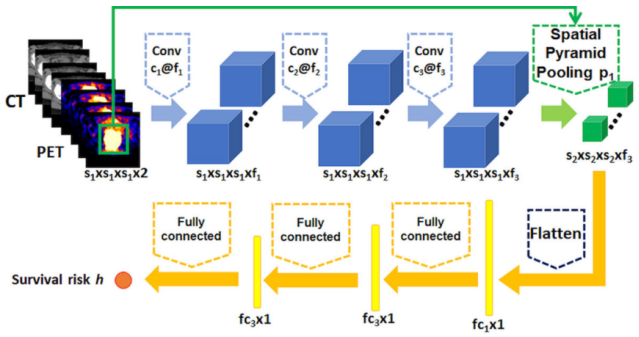


Figure 4.3: Rectal cancer dataset's model architecture
Reproduced from [43]

The last model [44] presented is a CNN model that was trained on CT images [1], they used 3 dataset of respectively 422, 68 and 30 images with the last dataset used for testing. The loss function used is also Equation 4.1. The model used is a very deep CNN with 6 convolutional layers and 3 max pooling layers with batch normalisation layers to help with the internal covariate shift problem that is caused by saturating non-linearities [19] (see subsection 1.3.6). The last part of the model is one fully connected layer and an output layer. They also used after the pooling layer, a dropout layer to avoid overfitting [47].

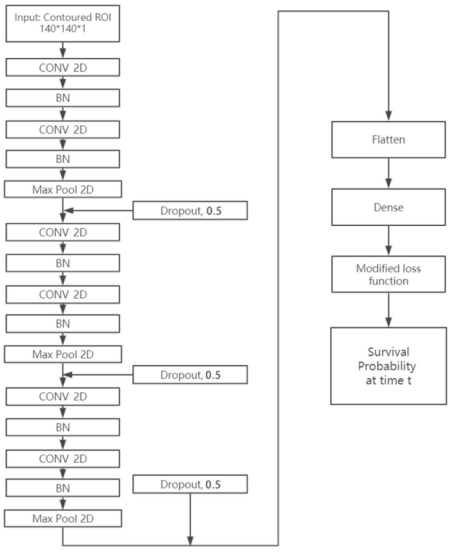


Figure 4.4: CT images' model architecture
Reproduced from [44]

The amount of censored patients is not notified in the three articles except for the generated stimulated dataset of Li and co. article [43].

[1] Computed Tomography scan, it is a medical imaging procedure that uses X-rays to produce detailed cross-sectional images of the body's internal structures.

In conclusion, CNN models can be used in the context of cytometry data taking into account that the kernel filter should have a shape that is not standard. It is also possible to modify the behaviour of neural networks to accommodate for survival data. Based on the CPH model, it is possible to derive a loss function that can be used to best predict risk scores. The challenge now is to combine those two principles in one single model.

Chapter 5

Model and experiments

5.1 Experiment protocol

5.1.1 Data

As detailed in chapter 3, addressing specific concerns related to the dataset is crucial. This ensures that the subsequent steps of the study are well-founded and effective.

One prominent issue stems from the size of the dataset. Model training is set to occur on Google servers, chosen for their superior RAM and GPU resources compared to typical personal computers (which are two key components for deep neural network training). Despite the server's robust capacities, it's crucial to consider their limitations in memory and GPU capabilities. Indeed, the dataset sheer volume makes it unfeasible to load the entire dataset into memory at once. Furthermore, the architectural compatibility between the model and the data dimensions necessitates attention, as storing extensive tensors in GPU memory is not feasible. While employing batching might offer a solution to manage memory usage in some cases, in our specific case, a limitation arises (which is discussed in detail in subsection 5.1.4), in that all subjects must be incorporated concurrently, rendering batch utilization unfeasible. As a result, the most feasible option is to choose a subset of cells that can be accommodated within memory without exacerbating concerns regarding the model's size.
The next problem to tackle is that not all patients have the same number of cells in their blood samples. This becomes tricky because the way convolutional layers work demands a fixed input size. Changing this size for every patient while using all patients during each training round is not feasible. To overcome this, we need to settle on a set number of cells.
This brings up a new question: what should we do about patients with the fewest cells, for example, one patient has 11,821 cells in their stimulated sample? If we stick to using just 22,000 cells for each patient (11,000 for each of the stimulated and unstimulated samples), it might not capture enough information for those with substantially more cells, say 200,000 in both samples. To address this, we decided to go beyond the 22,000 cell limit.
We tested out different cell counts – 50.000, 100.000, and 200.000 (going higher led to memory problems). While having more cells is better for calculations, it also takes

more time. So, if we get good results with fewer cells, that is worth considering.

For cell selection, a basic random sampling approach with replacement was employed. This strategy was chosen because cytometry data typically exhibits a connection between the presence or proportion of specific cell populations and the outcome [5]. Employing replacement during selection is anticipated to have minimal impact on the distribution of cell populations, ensuring that the inherent relationships between cells and outcomes remain largely unaltered. For consistency, an equivalent number of cells will be chosen for both stimulated and unstimulated samples.

Moving forward, the configuration of the input data necessitates consideration. The inquiry arises regarding the necessity of employing two input channels—one for stimulated cells and the other for unstimulated cells—or opting for concatenation or rearrangement. An insightful approach emerges from the FloReMi study [49], where features were computed for both stimulated and unstimulated cells, as well as the differences between them. This methodology implied that leveraging a convolutional model could be more effective if the weights applied to stimulated and unstimulated cells were identical, ensuring consistent feature computation for both types of cells. Consequently, utilizing two channels becomes superfluous if the same weights are employed across both channels. Shuffling the order of unstimulated and stimulated cells during input risks compromising the information associated with stimulation. As a resolution, the approach of concatenation was adopted, allowing for effective incorporation of relevant data.

The input data employed ultimately took the form of matrices with dimensions $n_{\text{cells}} \times n_{\text{markers}}$. Here, n_{cells} denotes the cumulative count of cells chosen from both the stimulated and unstimulated files, equally distributed between the two. Therefore, when referring to a test involving, for instance, 50,000 cells within this chapter, it signifies that 25,000 cells are stimulated, and the remaining 25,000 are unstimulated.

5.1.2 Loss function

The chosen loss function involves the negative log-likelihood, a modified version of the log-likelihood of the Cox Proportional Hazards (CPH) model. To reiterate for clarity, the mathematical expression is presented below:

$$\ell(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \delta_i \left[\mathbf{x}_i^\top \boldsymbol{\beta} - \log \left(\sum_{j \in \mathcal{R}_i} \exp(\mathbf{x}_j^\top \boldsymbol{\beta}) \right) \right] \tag{5.1}$$

where δ_i serves as a binary indicator, assuming the value of 1 if subject i experienced the event and 0 otherwise. Meanwhile, $\mathbf{x}_i^\top \boldsymbol{\beta}$ signifies the output of the network employing weights $\boldsymbol{\beta}$ for the input covariates $\mathbf{x}_i^\top$ associated with subject i.

There are some important points to consider:

- The loss function used here doesn't directly involve the outcome, which is the time it takes to progress to AIDS. Unlike most loss functions that measure the

difference between what a model predicts and what the true value is, this one is different. This function depends on the model's output for subject i (its risk score) and those for subjects j having a greater time to progression or a greater censoring time than subject i. This makes the loss function challenging to interpret, given that results can greatly vary depending on the distribution of events and censoring present in a dataset. It's crucial to note that if a model's loss function decreases, it indicates improved performance on the provided data which still holds for this loss function.

- The model only gives a risk score for each person, not an exact time to progression. So, these risk scores by themselves don't mean much. However, comparing them does make sense; higher risk scores should go along with shorter times to progression and vice versa.

- The value of the loss function depends on how many events (progressions) there are in the dataset. This means that if the validation set is smaller than the training set (which is common), the value for the validation set might be lower than for the training set, even if the model isn't performing as well on the validation set. To deal with this, we need to perform some adjustment (see 'Normalization' below).

- This loss function is at its lowest when the model's output helps to correctly sort patients by risk scores. The first term should contribute greatly in cases where the event has occurred. The second term serves as a normalization factor. As the event time progresses, fewer individuals remain at risk, and the second term will naturally decrease. This shows that there's a balance to strike while getting the patient classification right.

Normalization To facilitate the comparison between various datasets, a normalization process has been implemented for all presented results. It's important to note that $\sum_{i=1}^{n} \delta_i$ is greater when there are more events in the dataset, making it a suitable divisor to normalize across different sets.

Another significant consideration arises when dealing with imbalanced set sizes. For instance, if one set has four times more subjects compared to another, the average number of subjects j in the risk set of i will also be four times larger, thereby impacting the loss function's values. To address this, a decision was made: when the validation set is four times smaller than the training set, the data in the validation set will be counted four times in the loss computation.

Through these adjustments, the loss function can be effectively compared between datasets of different sizes.

Another noteworthy point is that the loss function provided to the model for feedback remains the one detailed in Equation 5.1, without normalization.

5.1.3 Concordance index based on inverse probability of censoring weighted (IPCW)

The concordance index offers a solution to the challenge of interpreting the loss function. This metric consistently ranges from 0 to 1, where 1 signifies a flawless model and 0.5 corresponds to random prediction [32]. However, an issue arises with the concordance index discussed in section 2.6; it is regarded as a biased estimator [50]. This bias tends to be overly optimistic under conditions of high censorship [50], [51].
To surmount this concern, a modified version of the concordance index has been proposed [50]. The Inverse Probability of Censoring Weighting (IPCW) approach seeks to rectify the potential bias introduced by censorship by assigning weights to each observation based on their estimated probability of being censored. This strategy assigns more weight to observations less likely to be censored and less weight to those more likely to be censored. Incorporating these weights into the IPCW concordance index yields a more accurate assessment of a survival model's predictive capability.
To compute this statistic, for each individual i at time t, it calculates the inverse probability of being censored weight, denoted as $W_i(t)$. This weight is usually computed using estimated censoring probabilities based on the Kaplan-Meier estimator fitted to the training data. The formula can be expressed as [50]:

$$W_i(t) = \frac{1}{P(\text{Censored}_i \leq t)} \tag{5.2}$$

Here, $P(\text{Censored}_i \leq t)$ represents the estimated probability that individual i is censored before or at time t. Subsequently, similar to the conventional concordance index, pairs of event/event or event/censored are compared and weighted using $W_i(t)$. Notably, the weight is higher for instances where there is a lower likelihood that individual i will be censored before or at time t, and vice versa.
Given the dataset's elevated proportion of censorship, this statistic will be employed instead of the traditional concordance index.

5.1.4 Cross-Validation, Folds, and Batch Size

Cross-Validation For evaluating our model, we opted for K-fold cross-validation is it is easy to comprehend, suitable for smaller data sets, and due to its capacity to provide less biased evaluations compared to single split methods. This technique partitions the dataset into k folds, employing each fold as a validation set while the remainder constitutes the training set. This allows to test the model k times, obtaining an averaged performance measure. The emphasis is placed on achieving consistent performance across multiple dataset splits, rather than excelling on a single split. This approach serves to evaluate the model's generalization capabilities and its susceptibility to overfitting [45]. The choice was made

Fold Size Our decision was to employ a 4-fold cross-validation protocol, aligning with the prevalent 75-25% data split commonly used in machine learning. This choice balances the advantage of numerous folds for thorough testing with the need to maintain an adequate sample size in the validation set. This prevents excessive variability

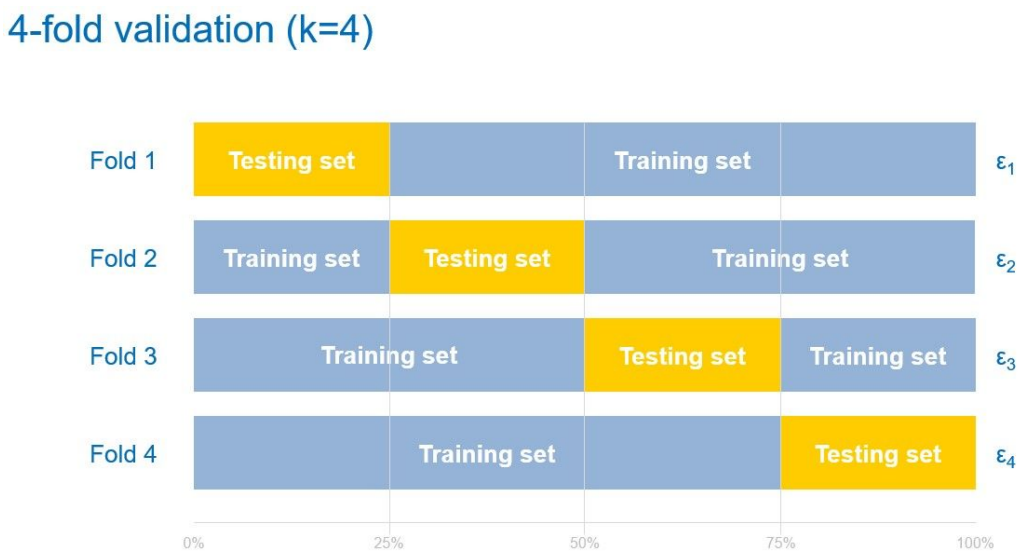


Figure 5.1: Illustration of cross-validation with 4 folds. Reproduced from [52].

in the concordance index and loss function arising from a scarcity of events. On average, each validation set is expected to comprise $34/4 = 8.5$ events.

Fold Creation The computation of the loss function and c-index depends heavily on event distribution and the prevalence of censored samples with $t_{\text{censored}} > t_{\text{event}}$ for specific events. To mitigate convergence challenges and facilitate effective model training, manual splits are introduced. The subjects are stratified into event and censored samples, each further sub-stratified into 10 time quantiles (deciles [53]), both event and censored subjects are randomly and equally assigned to folds based on these 20 different strata.
For instance, in the dataset, there are 4 events in the first decile. The objective is to evenly distribute one event across each of the 4 folds created. With 16 censored samples in the first decile, each fold should incorporate an additional 4 samples. This procedure is replicated across all deciles for both events and censored samples, culminating in the establishment of the 4 folds. This approach ensures uniform distribution of event and censored subjects while maintaining a consistent time distribution within each fold.

Batch Size Similar reasoning underpins the determination of the batch size. To accurately compute the loss function, it is essential to maximize the number of subjects in the risk set of each event. Consequently, all samples in the training set are processed simultaneously.

5.1.5 Learning Rate, Epochs, and Optimizer

Epochs The training process was conducted over 1000 epochs for all experiments. The utilization of a high number of epochs stems from the model's relatively slow convergence, which will be demonstrated in subsequent experiments.

Learning Rate A learning rate of 10^{-3} was employed. It was made to obtain a balance between quick convergence and model stability. The latter is crucial, as high learning rates can compromise stability [54].

Optimizer The chosen optimizer is the Adam optimizer, a widely acclaimed algorithm that surpasses the traditional stochastic gradient descent [55].

5.2 Base Model

Based on the articles presented in the preceding chapter, an initial model was designed. The architectural structure of this model is showed in figure 5.2. This section will focus on the major components of the model and explain the reasons for our choices.

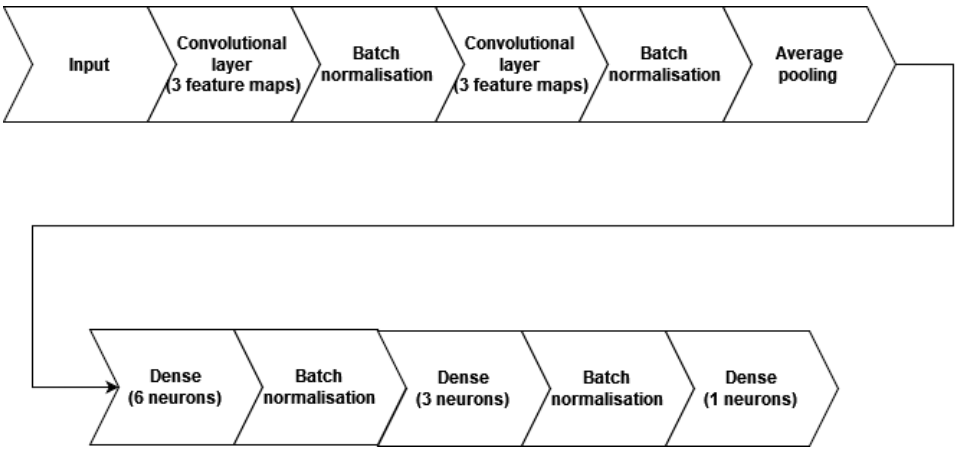


Figure 5.2: Schematic representation of the Base Model.

5.2.1 Convolutional Layers

To begin with, we opt a simpler model due to its ease of training and understanding. If it turns out to be insufficient for the data after testing, adjustments can be made. Ultimately, a simpler model that performs as well as a more complex one is preferable. More complex models also tend to overfit more [56].
The model initiates with 2 convolutional layers resembling those discussed in section 4.2. The input data is processed by the first convolutional layer across the marker dimension using a ReLU activation function. The resulting outputs from this layer are three tensors with dimensions $n_{\text{cells}} \times 1$, whereas the initial input data has dimensions $n_{\text{cells}} \times n_{\text{markers}}$.
As illustrated in figure 5.3, the second convolutional layer computes a linear combination of the three values from the output tensors of the first layer that are on the same row. Similarly, it produces three tensors with dimensions matching those of the output tensors of the first convolutional layer. The activation function for this layer is also ReLU.

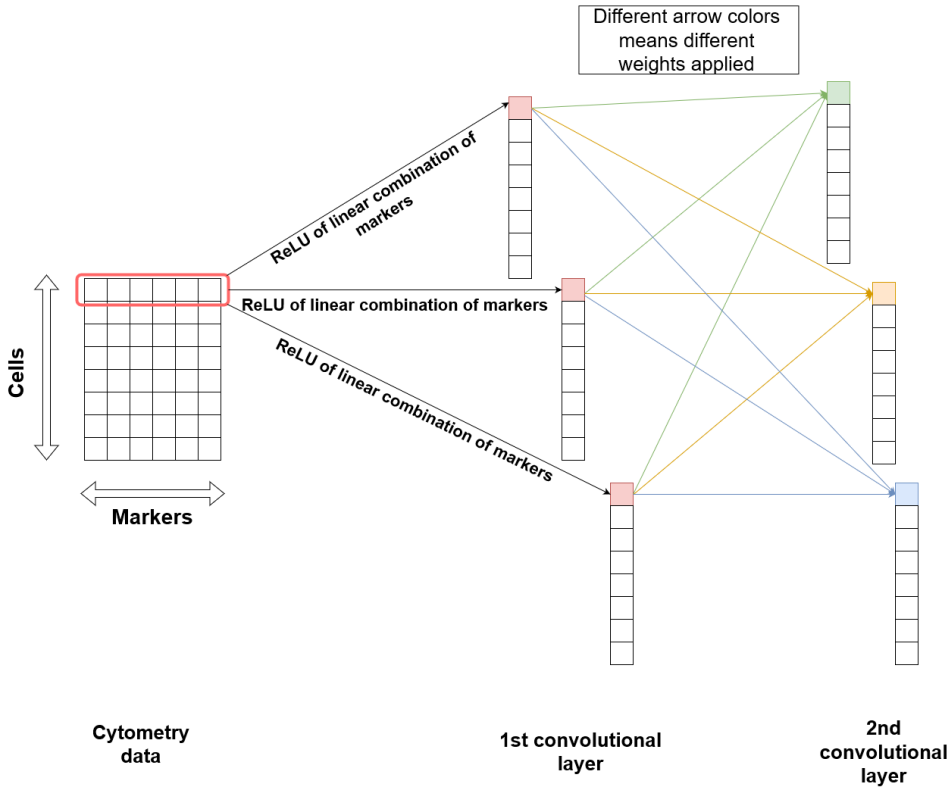


Figure 5.3: Schematic representation of the first two layers of the base model.

This configuration showed promising results with CyTOF data [5]. Therefore, the aim was to determine if this structure serves as a solid starting point for the data in this thesis. Furthermore, this model is not overly complex and aligns well with the nature of cytometry data. The choice of activation functions is grounded in their proximity to linear functions while retaining non-linear properties, making them suitable starting points. The ReLU activation function is particularly popular for internal layers because it helps tackling the issue of vanishing gradient [20].

5.2.2 Pooling

Following the two convolutional layers, a pooling layer is introduced. This transition is necessary to shift from cellular-level features to subject-level features. As depicted in figure 5.4, given that the stimulated and unstimulated cells are concatenated, the pooling process is divided into two parts: pooling for the stimulated cells and pooling for the unstimulated cells. This division enables the creation of subject-level features for both cell types, facilitating subsequent operations as demonstrated by the FloReMi team [49].

We chose the average pooling layer due to the dependency of the outcome on the proportions of specific cell populations [3]. A max pooling approach would not adequately represent this dependency. Max pooling selects the cells with the highest values after convolution without considering proportions.

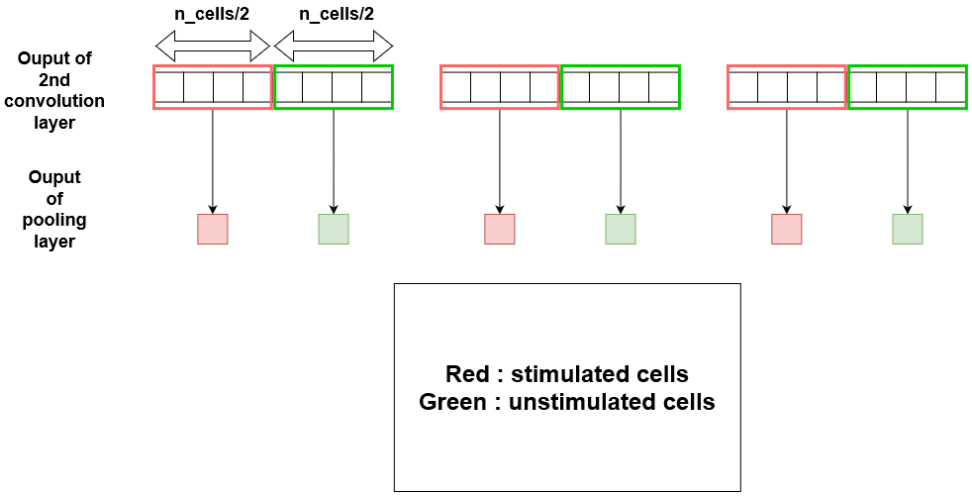


Figure 5.4: Schematic representation of the pooling layer.

5.2.3 Dense Layers

After the pooling layers, two dense layers and an output layer are incorporated. The first dense layer comprises 6 neurons, the second has 3, and the output layer consists of a single neuron. Activation functions for the initial two dense layers are ReLU, providing non-linearities, while the output layer uses a linear activation function to distinguish negative outcomes.

5.2.4 Batch Normalisation Layers

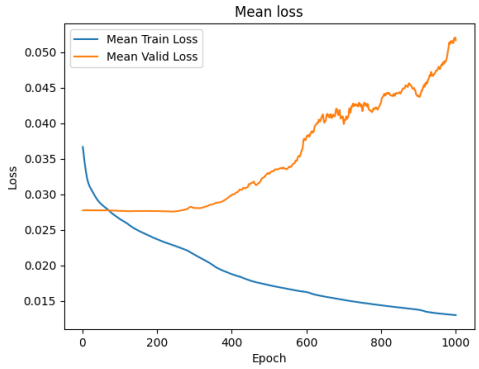
To enhance training stability and convergence, batch normalisation layers are introduced. In the initial phase of this study, we encountered training challenges where specific folds failed to converge, leading to a stuck concordance index of 0.5 and a constant loss function value. This situation was notably affected by weight initialization across layers, as repeating the process on the same fold occasionally resulted in convergence. Leveraging its stabilizing effect, batch normalization proved important in mitigating this concern. One batch normalisation layer follows each layer except for the input, pooling, and output layers.

5.3 Evaluation of the Base Model

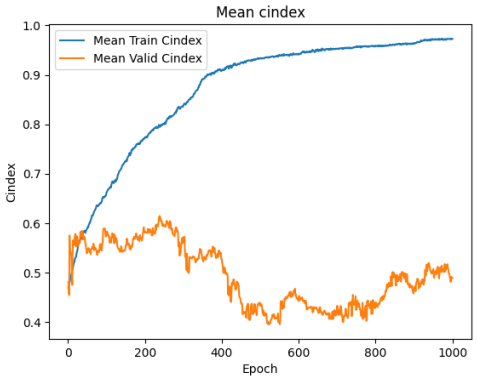
In this section, we present the results of our initial model experimentation. As discussed in section 5.1, the experiments were conducted using 50.000, 100.000, and 200.000 randomly selected total cells. The graphs presented here depict either the progression of the concordance index or the loss function.

The performance of the base model with different number of cells is illustrated in figure 5.5. Several observations can be obtained from the graphs:

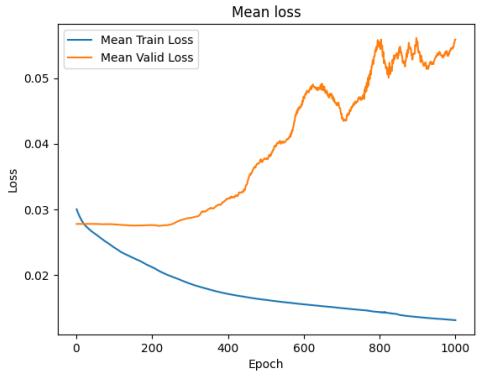
- The loss function of the validation set tends to rise over time, indicating over-



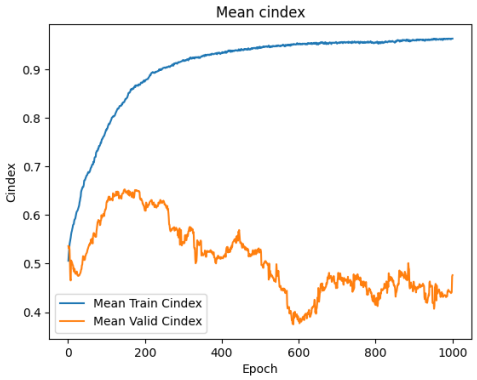
(a) Mean loss across the 4 folds with 50000 total cells selected.



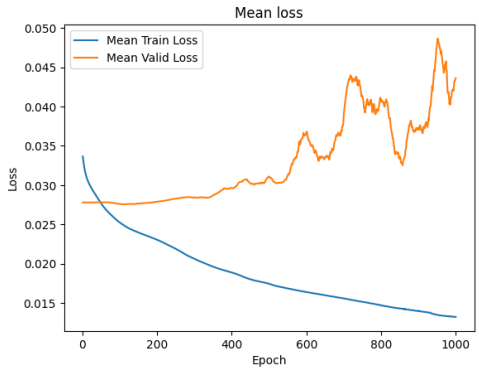
(b) Mean cindex across the 4 folds with 50000 total cells selected.



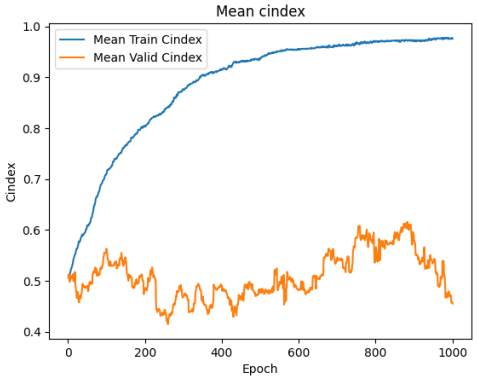
(c) Mean loss across the 4 folds with 100000 total cells selected.



(d) Mean cindex across the 4 folds with 100000 total cells selected.



(e) Mean loss across the 4 folds with 200000 total cells selected.



(f) Mean cindex across the 4 folds with 200000 total cells selected.

Figure 5.5: Performance of the base model with 50000, 100000 and 200000 total cells selected.

fitting. This trend is also reflected in the cindex plots, where the cindex of the training set approaches 1 while that of the validation set fluctuates around 0.5, indicating a random predictor.

- Regardless of the number of cells used, the results remain the same, showing no significant improvement.

- While the loss function of the training set descent becomes milder towards the end, it continues to decrease, signifying that the minimum has not yet been reached. This finding supports the notion that the model's convergence is slow.

Considering these observations, we propose potential actions for addressing the issues identified and propose experiments to test their impact:

- Overfitting could stem from a high learning rate [54]. To test this hypothesis, we can run the same model with a reduced learning rate. Notably, no differentiation in performance was seen across various cell counts. If the learning rate is indeed problematic, then it could have masked the potential improvements that more cells can bring.

- Overfitting can also be mitigated by employing L1 or L2 regularisation, introducing a term in the loss function related to network weights [57], or by incorporating dropout layers that cut connections between neurons in different layers [47]. These techniques are widely utilized to fight overfitting.

- Data augmentation, involving the introduction of noise or transformations to the data [48], is another strategy for fighting overfitting.

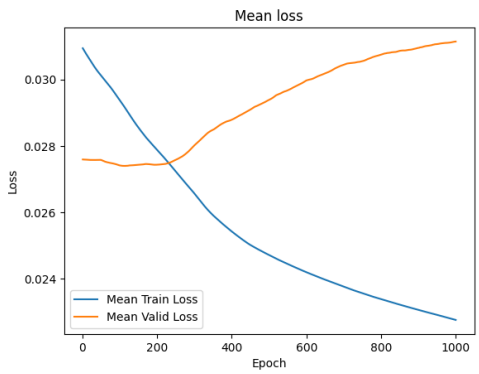
In the upcoming sections, we will test these approaches to determine which one yields improvements in performance.

5.4 Adjusting the Learning Rate

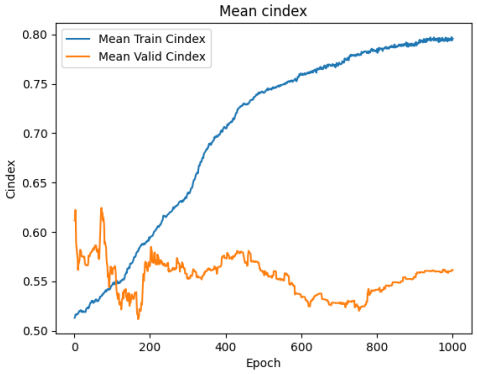
In this section, we investigate the lowering of the learning rate to assess its potential as a solution to the overfitting issue observed earlier.
Our experiment will involve a comparison of results obtained from the input datasets containing 50.000 and 200.000 total cells. The chosen learning rate for this exploration is set at 10^{-4} (compared to 10^{-3} in the initial experiment.
Our objectives for this investigation are twofold:

- Investigate if the learning rate can be the reason of the overfitting.

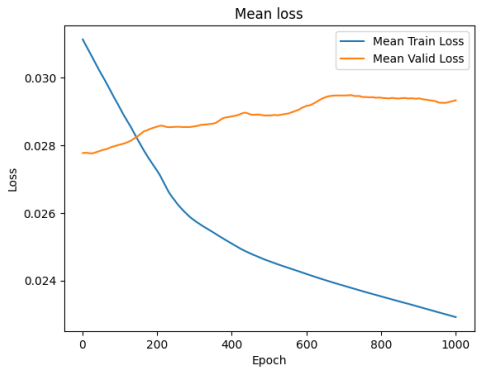
- Determine whether employing a lower learning rate yields improved model performance on datasets with a larger number of cells, in contrast to the findings from the previous section.



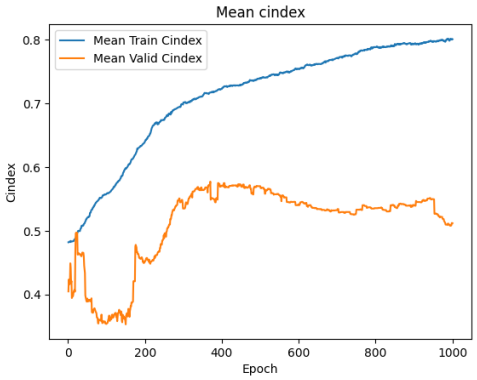
(a) Mean loss across the 4 folds with 50000 total cells selected.



(b) Mean cindex across the 4 folds with 50000 total cells selected.



(c) Mean loss across the 4 folds with 200000 total cells selected.



(d) Mean cindex across the 4 folds with 200000 total cells selected.

Figure 5.6: Performance of the base model with 50000 and 200000 total cells selected and a learning rate of 10^{-4}.

It's important to anticipate that with a decreased learning rate, the model's convergence on the training set is expected to be slower. Consequently, achieving similar performance levels on the training set, while employing the same number of epochs, is not the anticipated outcome.
The results depicted in figure 5.6 present a nuanced picture. While the adoption of a lower learning rate of 10^{-4} hasn't completely eliminated the overfitting problem, some positive effects are observed. For instance, in the case of the 200.000 cell dataset, the loss function of the validation set displays a flatter trajectory, and the cindex shows increased stability, indicating a reduction in overfitting effects. However, the validation loss function is still increasing, while the training set loss function continues to decrease. As predicted, the performance of the training set within 1000 epochs is not as high as the previous outcomes, yet the loss function's downward trend suggests the potential for better training set performance with additional epochs. In conclusion, the adjustment of the magnitude of weight updates does not appear to significantly influence the overfitting issue. As a result, we will continue employing a learning rate of 10^{-3} for subsequent tests.

5.5 Regularisation and Dropout for Overfitting Mitigation

In this section, we experiment using regularisation and dropout layers, aiming to mitigate the overfitting issues that have been identified. We will begin by providing a brief explanation of these techniques before presenting the experimental results.

5.5.1 Regularisation

Regularisation methods are employed to counteract overfitting problems by introducing terms into the loss function that penalize model complexity. Within this realm, we explore two specific types: L1 and L2 regularisation.

L1 Regularisation This approach augments the loss function with a term proportional to the sum of the absolute values of the model weights. The mathematical expression for L1 regularisation is as follows [58]:

$$\text{Modified Loss}_{L1} = \text{Loss} + \lambda_1 \sum_{i=1}^{N} |w_i| \tag{5.3}$$

Here, λ_1 represents a parameter that governs the strength of the regularisation effect. The L1 regularisation encourages less influential weights to be set to zero, thus promoting a sparser network structure.

L2 Regularisation In contrast, L2 regularisation augments the loss function with a term proportional to the sum of the squared norms of the weights [58]:

$$\text{Modified Loss}_{L2} = \text{Loss} + \lambda_2 \sum_{i=1}^{N} \|w_i\|^2 \tag{5.4}$$

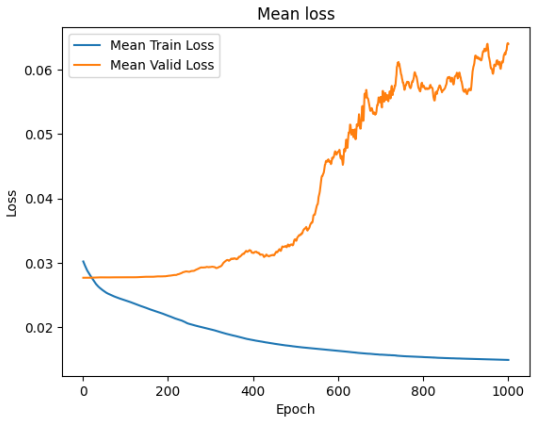
where λ_2 represents a parameter that dictate the strength of the L2 regularisation. While L2 also penalises weight magnitudes, it doesn't force them to zero. Instead, it results in smaller weight values, promoting a network with reduced sensitivities

If L2 regularisation outperforms L1, it signifies that correlated features with varying degrees of importance contribute collectively to overfitting, and L2's smooth shrinkage aids in achieving better generalization by balancing these contributions. In contrast, L1's superiority suggests that many features are irrelevant, and its sparsity-inducing nature effectively selects the most informative variables, mitigating overfitting by simplifying the model [57].

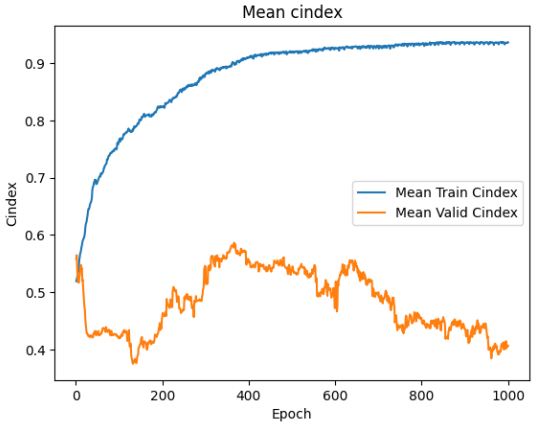
Results with regularisation

As a continuation of our previous discussions, we observe that there appears to be minimal variation in model performance across different numbers of cells. Building upon this insight, we proceed to evaluate the model using a learning rate of 10^{-3}, and we opt to start our experiments with a dataset of 50.000 total cells. Should we witness performance improvements, we can then consider expanding our evaluation to take into account larger input datasets. It is important to note that regularisation techniques are incorporated into all layers, except for the pooling layer, as it does not possess any trainable parameters.
To systematically explore the effects of regularization, we introduce two distinct parameter values, λ_1 and λ_2, set at 0.01, a standard value for L1/L2 regularization, along with 1 and 100. This exploration is designed to verify whether the use of weight penalties, particularly at a substantial magnitude, can indeed mitigate the overfitting.

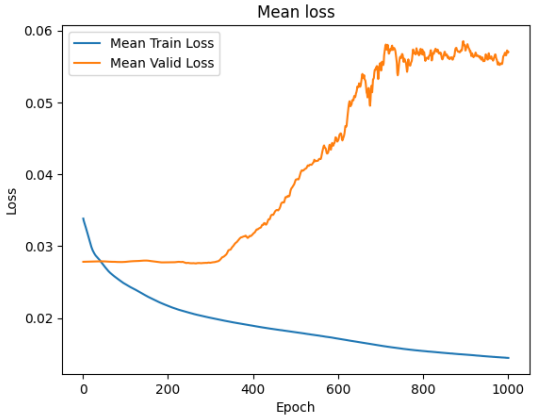
Upon conducting the experiments, it becomes apparent that the introduction of regularisation does not yield substantial improvements in model performance, as illustrated in figure 5.7 and figure 5.8. L1 regularisation, intended to push less significant weights to zero, doesn't show notable benefits, potentially suggesting that the problem of overfitting does not originate from irrelevant features. On the other hand, L2 regularisation, despite promoting smaller weight values, does not seem to have an impact on the overfitting issue significantly.



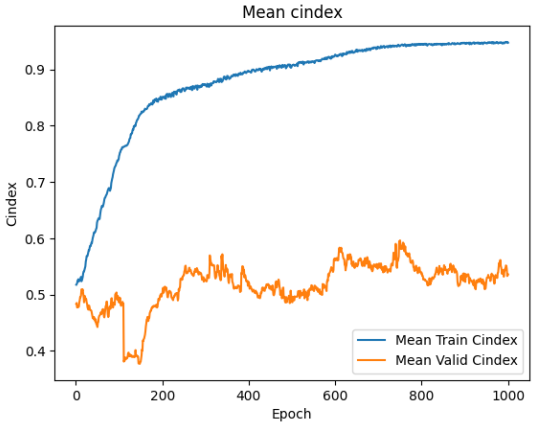
(a) Mean loss on 4 folds with $\lambda_1 = 0.01$.



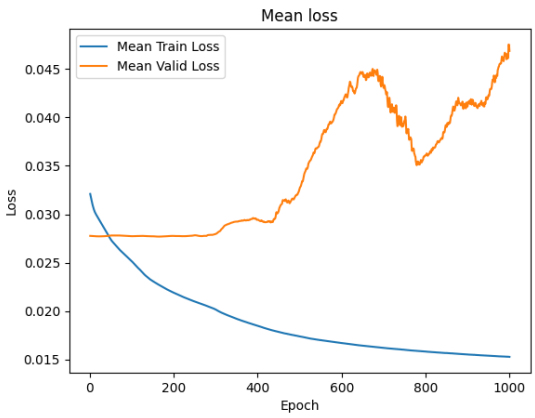
(b) Mean cindex on 4 folds with $\lambda_1 = 0.01$.



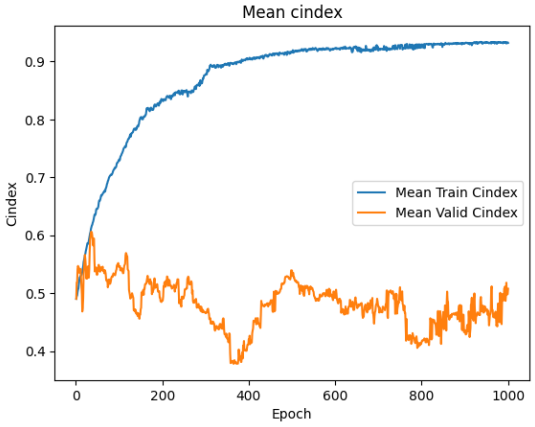
(c) Mean loss on 4 folds with $\lambda_1 = 1$.



(d) Mean cindex on 4 folds with $\lambda_1 = 1$.

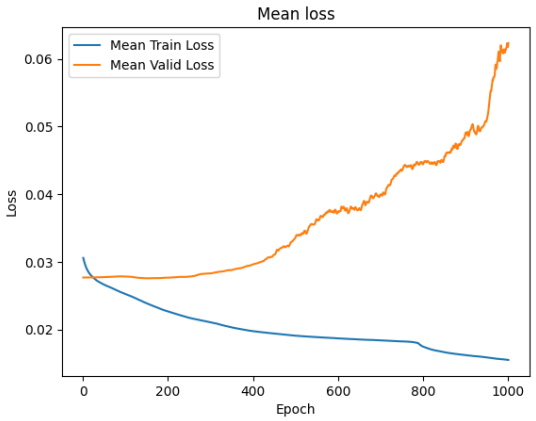


(e) Mean loss on 4 folds with $\lambda_1 = 100$.

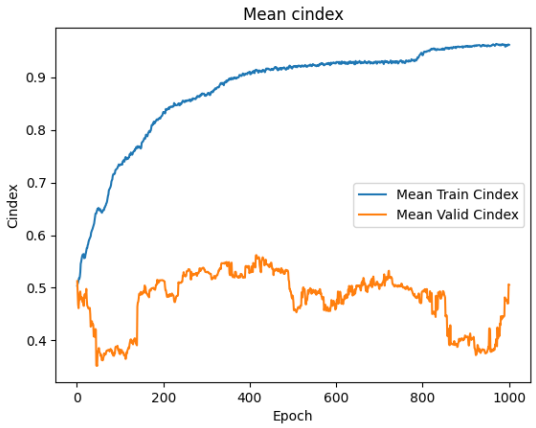


(f) Mean cindex on 4 folds with $\lambda_1 = 100$.

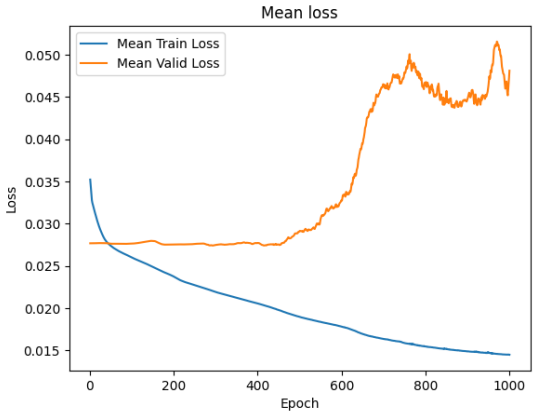
Figure 5.7: Performance of the base model with L1 regularisation.



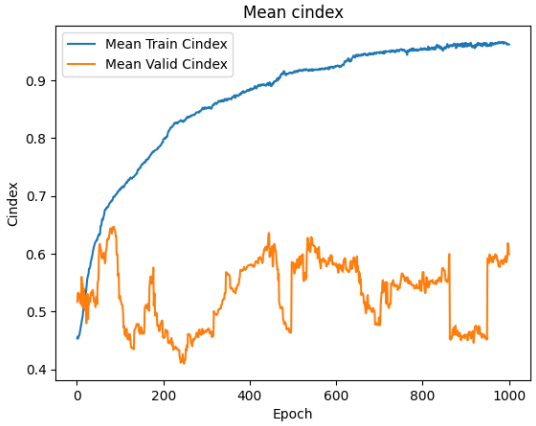
(a) Mean loss on 4 folds with $\lambda_2 = 0.01$.



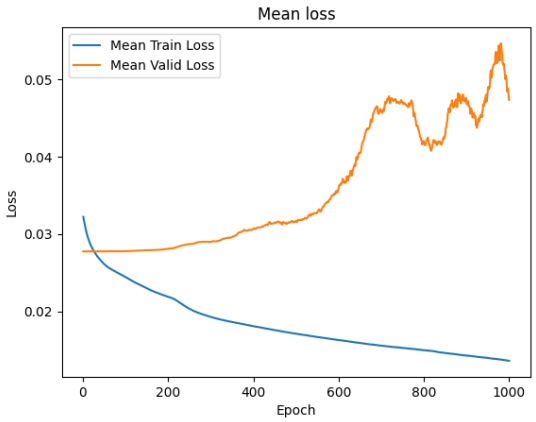
(b) Mean cindex on 4 folds with $\lambda_2 = 0.01$.



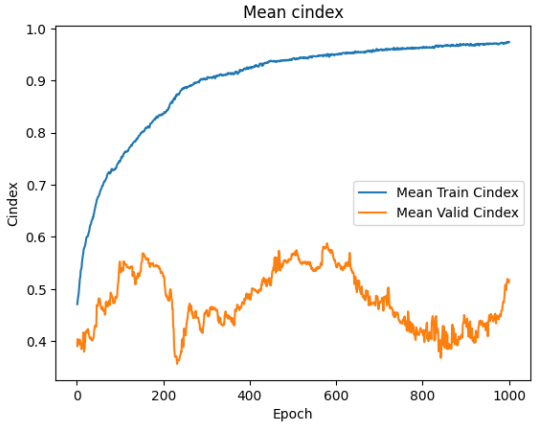
(c) Mean loss on 4 folds with $\lambda_2 = 1$.



(d) Mean cindex on 4 folds with $\lambda_2 = 1$.



(e) Mean loss on 4 folds with $\lambda_2 = 100$.

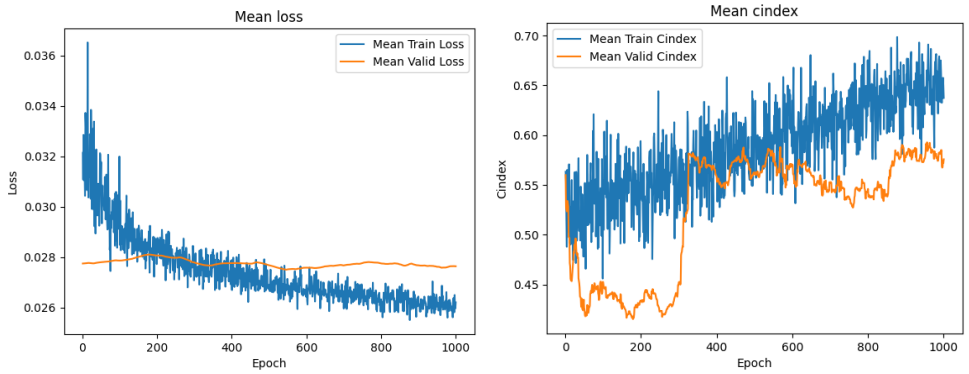


(f) Mean cindex on 4 folds with $\lambda_2 = 100$.

Figure 5.8: Performance of the base model with L2 regularisation.

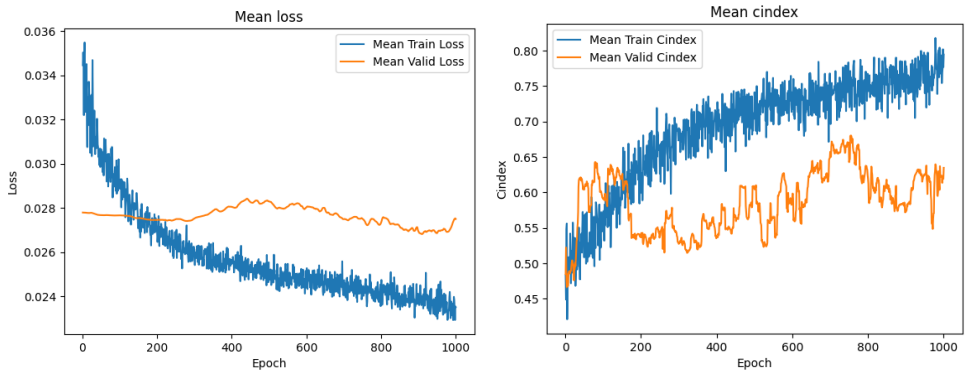
5.5.2 Dropout layer

Dropout layers are a technique to address overfitting by randomly deactivating connections between neurons during training. This approach encourages neurons to generalize better, as they cannot rely on specific connections for accurate predictions [47]. These layers have a parameter, the dropout probability or rate which defines what is the probability that a link is dropped out between two layers. Thus for an infinite amount of epochs, one would convince himself that in average the amount of links that will be dropped is the total number of links multiplied by the dropout rate [47]. We conducted experiments with different dropout rates, including the standard values of 0.5 for hidden layers and 0.2 for the input layer [18], as well as alternative values such as 0.25 for hidden layers and 0.1 for the input layer. The dropout layers were incorporated after each convolutional and dense layer, except for the output layer.



(a) Mean loss across the 4 folds with dropout layers (rate are 0.5 and 0.2).

(b) Mean cindex across the 4 folds with dropout layers (rate are 0.5 and 0.2).



(c) Mean loss across the 4 folds with dropout layers (rate are 0.25 and 0.1).

(d) Mean cindex across the 4 folds with dropout layers (rate are 0.25 and 0.1).

Figure 5.9: Performance of the base model with dropout layers.

The results, as displayed in figure 5.9, indicate that while dropout layers do not entirely eliminate overfitting, they have a positive impact on model performance. In particular, for dropout rates of 0.5 for hidden layers and 0.2 for the input layer, the validation loss and cindex remain more stable, with the cindex reaching levels closer to 0.6. These findings suggest that allowing neurons to specialize less by introducing

dropout layers can lead to improved generalization capabilities.

In conclusion, while the impact of regularisation seems limited in addressing overfitting, incorporating dropout layers offers a positive step towards mitigating the overfitting problem. Therefore, dropout layers will be retained in the subsequent iterations of the model tests.

5.6 Data Augmentation

Data augmentation is a technique employed in neural networks to address overfitting. This approach involves generating modified instances of the original data. Modifications can include introducing noise, applying geometric transformations (for image data), or other alterations. The primary goal is to expose the model to slightly different input data while preserving the same desired outcomes. This strategy helps prevent the model from learning overly specific patterns from the training data, thereby enhancing its ability to generalize [48], [59].
Given the unique nature of our dataset, implementing data augmentation on the input data is challenging. Determining appropriate transformations for this data type requires significant effort and could potentially introduce unintended artifacts. Noising the data is also complex due to the variability of cell markers, making a generalized noise addition impractical.
Consequently, the focus has shifted to augmenting censoring/event times. By adding controlled perturbations to the timing, the aim is to introduce small variations while maintaining data integrity. The foundational study [3] indicates that time values are expressed in days, suggesting a precision of around 1 day. To mimic realistic variability, Gaussian noise with a mean of 0 days and a standard deviation of half a day was applied. This choice ensures that roughly 68% of the time modifications fall within a half-day range, while 95% remain within 1 day [60].
Building on insights from previous sections, it is evident that the model could benefit from additional data, especially censored data to enhance generalization. As a result, it was decided to generate 809 virtual subjects in addition to the existing 191 real subjects, resulting in a total of 1000 subjects.
The procedure for generating virtual subjects is as follows:

1. Randomly select an existing study subject using a uniform distribution.

2. Preserve cytometry data while introducing Gaussian noise with a mean of 0 and a standard deviation of 0.5 days to event/censoring times.

3. Iterate this process to create 809 virtual subjects.

While this procedure doesn't guarantee exact distribution parity, it approximates the real distribution. The histogram of times for real and augmented data, shown in figure 5.10, illustrates their striking similarity, reinforcing the authenticity of augmented data.
To evaluate data augmentation, a c-index curve for real data is incorporated into the validation set evaluation. This design retains the inclusion of dropout layers, which

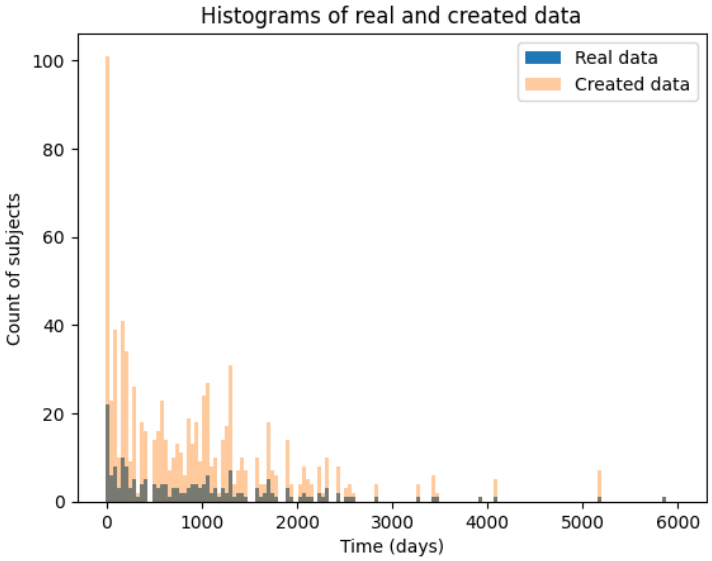
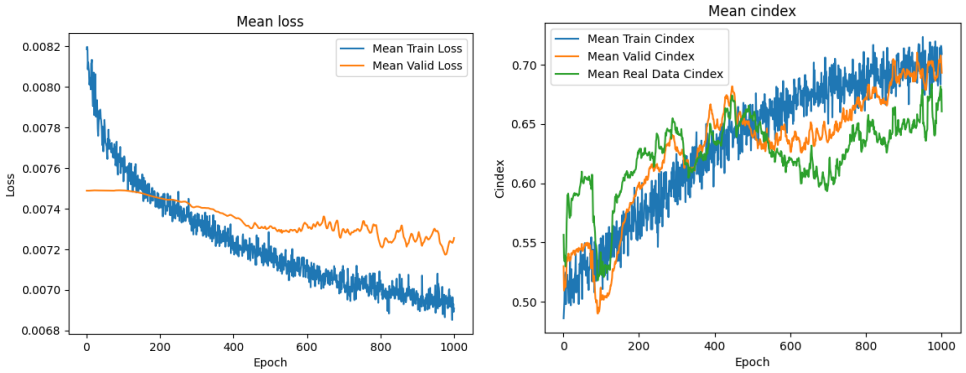


Figure 5.10: Histogram of the real and created data times (censored and events aggregated).

previously demonstrated positive impacts on validation set performance. Additionally, due to GPU memory limitations, we had to reduce the number of total cells per patient to 20.000 instead of 50.000. This change was necessary because the increase in the number of patients contributed to this limitation.



(a) Mean loss across the 4 folds.

(b) Mean cindex across the 4 folds.

Figure 5.11: Performance of the base model on the augmented data set.

Figure 5.11 presents the performance outcomes of the augmented dataset:

- The validation set loss function exhibits a notable trend toward convergence. Although the rate of descent is less pronounced than the training loss, it marks the first instance of concurrence between training and validation losses.
- While the training set c-index after 1000 epochs is lower compared to previous experiments (except for the dropout scenario), a significant reduction in overfitting is evident. The orange curve and the blue curve reach similar levels after 1000 epochs.

- Data augmentation yields promising results. The green c-index curve, representing the model's performance on real data within the validation set, closely aligns with the overall validation set's c-index.

- Data augmentation contributes to improved model generalization. Exposure to nearly identical cytometry data linked to slightly varied event times enhances the model's grasp of underlying relationships.

5.7 Feature Maps

A possible next step for enhancing model performance that remains unexplored is the increase of the number of feature maps in the convolutional layers.

5.7.1 Motivation

As stated in section 3.4, bioinformaticians utilize a gating technique to establish connections between cytometry data and outcomes. This approach involves filtering through markers to identify cell populations linked to the outcome, with typical studies identifying 20-30 distinct cell populations, as showed in figure 5.12 (highlighting 16 populations).

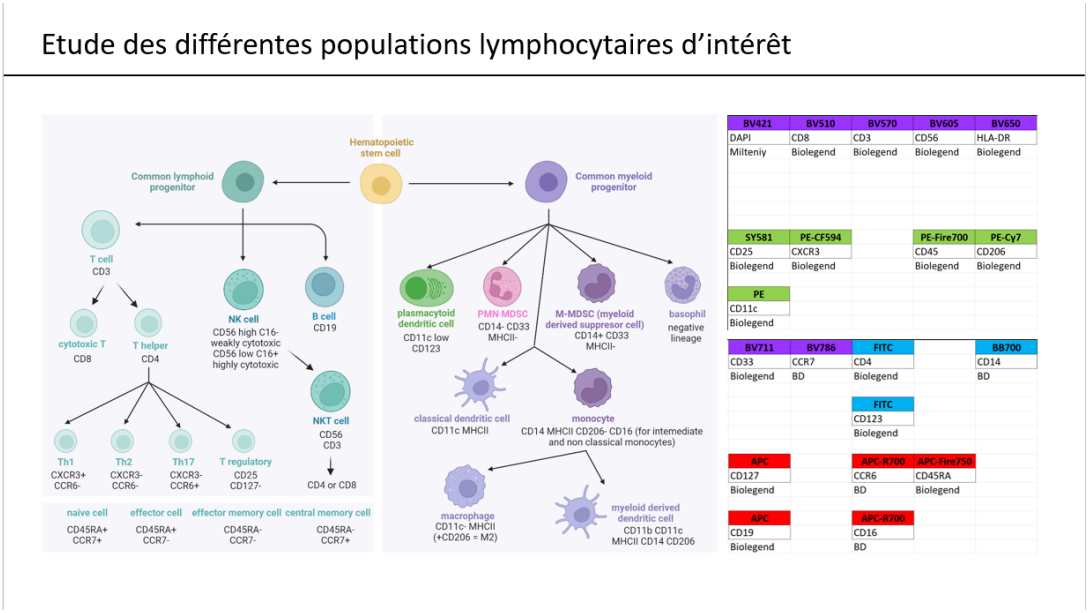


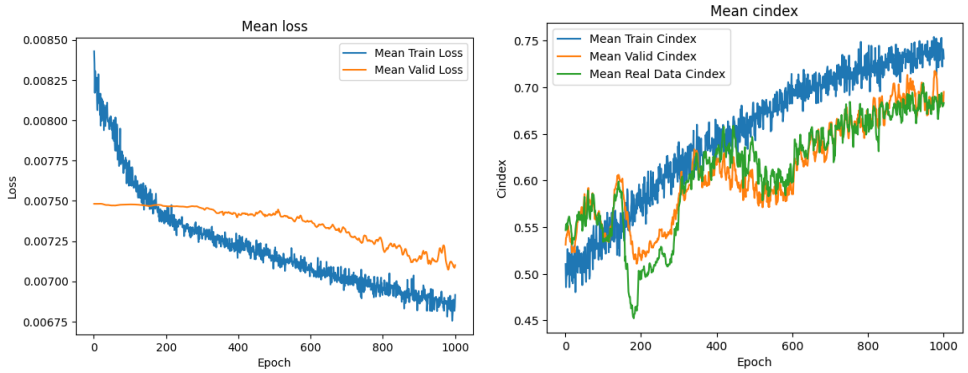
Figure 5.12: Investigation of different lymphocyte populations of interest. Shared by M. Hauchamps. Pictures were designed on BioRender software[1].

Assuming that our model treats cytometry data analogously, considering every marker as either on or off (though this simplification is not accurate), our present model can merely represent $2^3 = 8$ potential cell populations (3 feature maps based on on and

[1] https://www.biorender.com/

off markers). This limitation is inadequate for capturing the complexity of most studies. Consequently, an argument arises for increasing the number of feature maps to enable the model to identify the full spectrum of potential cell populations. In pursuit of this, we intend to examine a model with 6 feature maps, theoretically accommodating $2^6 = 64$ distinct cell populations. Accordingly, the convolutional layers will be modified from 3 feature maps to 6 feature maps.

5.7.2 Results



(a) Mean loss across the 4 folds. (b) Mean cindex across the 4 folds.

Figure 5.13: Performance of the model with 6 feature maps.

Inspecting the c-index graph, we observe a modest enhancement in the validation set (approximately 0.7), accompanied by a slight improvement in the training set (formerly around 0.7, now nearing 0.75). This pattern suggests that the existing feature map size was adequate for translating the information from markers into higher-level features. Furthermore, the results imply that extending the training duration could lead to further performance gains.

5.8 Performance on the Test Set

The performance of the best model is evaluated on the provided test set of the FlowCAP IV challenge [3]. The utilized model consists of two convolutional layers with 6 feature maps, an average pooling layer, and 3 dense layers with 6, 3, and 1 neurons respectively. Each layer, except for the pooling and output layers, is followed by a dropout layer with a rate of 0.2 for the input layer and 0.5 for the others. Additionally, batch normalization layers are employed after each convolutional and dense layer except the ouput layer.
The training data are the augmented dataset, generated from the 191 real training subjects, yielding a total of 1000 subjects. For each subject in both the test and training sets, 10.000 stimulated and 10.000 unstimulated cells are randomly selected. The test set is made of the remaining 192 subjects from the study.

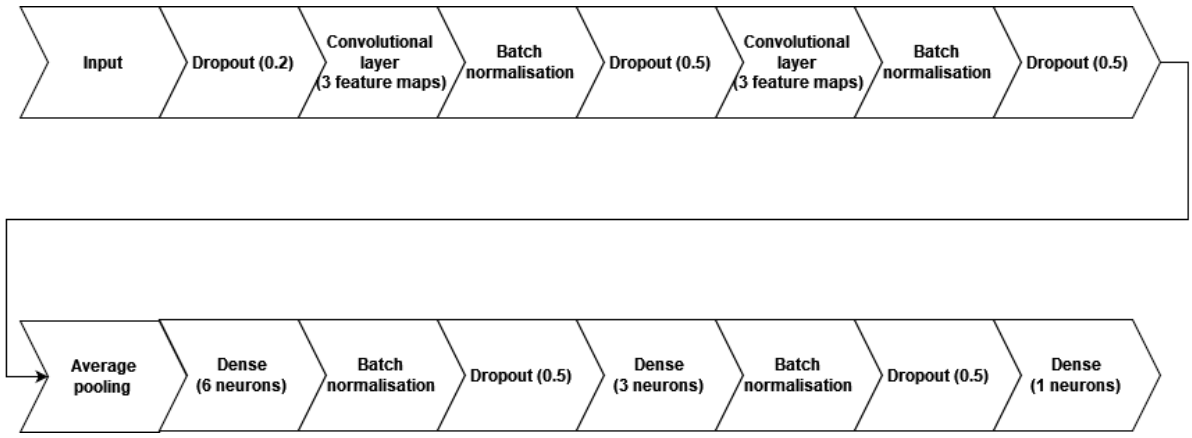
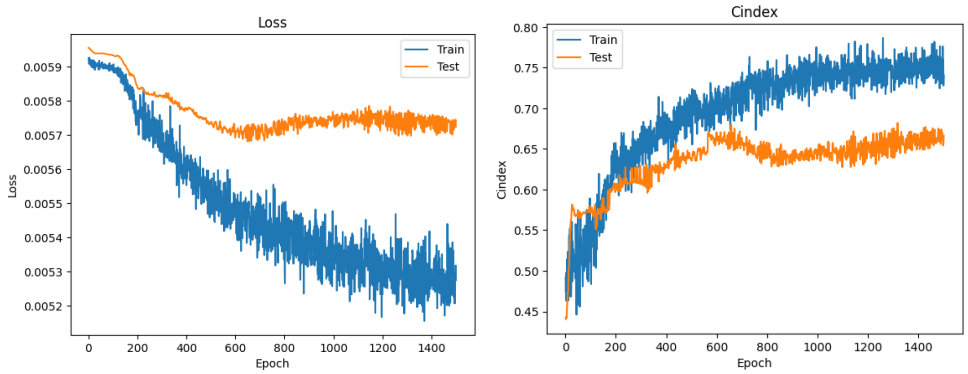


Figure 5.14: Architecture of the best-performing model.

The model is trained for 1500 epochs, considering the pattern observed in previous experiments, where the loss function continued to decrease after 1000 epochs. The learning rate employed is 10^{-3}.



(a) Loss evolution for training and test sets.

(b) C-Index evolution for training and test sets.

Figure 5.15: Performance of the final model on training (blue) and test (orange) sets.

Several observations emerge from the results:

- The model could potentially benefit from further training, as the training loss has not yet plateaued. Despite this, the downward trajectory of the loss and the upward trend of the c-index have slowed considerably.
- Within the first 600 epochs, the test loss experiences a decline. Subsequently, a plateau is reached, followed by a slower reduction after 1200 epochs. Similar trends are observed in the c-index, with an initial rise, followed by a plateau, and another increase. Extending the training duration might yield further performance improvements.
- The c-index demonstrates that overfitting is not a significant concern, given that the training c-index hovers around 0.75, while the test c-index remains at approximately 0.65.

The model's performance on the test set after 1500 epochs yields a c-index of 0.666 with a 95% confidence interval of $[0.599, 0.733]$ [61], closely resembling the performance of the FloReMi team's random forest model, which achieved a c-index of 0.672 [49]. This outcome is promising, hinting at the potential for greater performance gains through alternative methods or further fine-tuning of the current model.

Chapter 6

Conclusion

In this thesis, we set out to create a deep learning model to predict the risks linked to the time to AIDS progression in HIV-positive patients, using flow cytometry data provided by the FlowCAP consortium [3]. The dataset contained 383 subjects, each with 2 blood samples taken (stimulated in vitro with HIV antigens and unstimulated). However, only 191 subjects formed the available training data, while the remaining 192 served as a testing set. Throughout our exploration, we navigated through stages of model development, testing, and refining, uncovering valuable insights into the intricacies of the task.

The challenge was substantial, combining deep neural networks, survival analysis, and flow cytometry data. Our dataset featured many censored subjects, making the task even more complex. Furthermore, we grappled with a novel loss function that proved challenging to interpret, a model designed for different data, and the need for an alternative metric to assess our model performance. The variability in cytometry data size posed an additional challenge, with each subject's data containing different cell counts and having a high memory need.

Given these challenges, we built a base model inspired by the CyTOF model [5], using convolutional neural networks (CNNs). To accommodate survival data, we adjusted the partial log-likelihood of the Cox Proportional Hazards model. Our initial experiments exposed the difficulties of overfitting and prompted us to explore solutions.

While regularization techniques like L1 and L2 were aimed at controlling model complexity, they didn't yield improvements for our task. Subsequently, we introduced dropout layers, which alleviated overfitting and led to modest enhancements. Despite this, our validation loss seemed to plateau, and model performance on the test set remained stagnant.

In an effort to enhance performance further, we experimented with data augmentation. By introducing Gaussian noise to event/censored times, we artificially expanded our dataset to include 809 "virtual" subjects, totalling 1000. This approach effectively curbed overfitting and produced the first instance of decreasing loss on the validation set. It highlighted the need for more diverse examples in our original dataset and con-

firmed the effectiveness of data augmentation.

Although our methods yielded improved outcomes, our model's performance remained unsatisfactory. Inspired by classical flow cytometry analyses, we attempted to address this by increasing the number of feature maps in our convolutional layers from 3 to 6 to express all the data' complexity, resulting in a minor performance boost.

Ultimately, we evaluated our best model against the test set of 192 subjects, comparing our results with the winning team of the challenge, FloReMi [49]. While our performance closely matched theirs, it did not surpass it.

Our journey demonstrated the potential of convolutional neural networks for tackling complex data analysis tasks beyond traditional image analysis. We successfully demonstrated the competitiveness of our model against established methods, with reduced manual computation. However, further exploration is warranted to assert whether our approach can outperform existing models. Future investigations could involve combining multiple proposed solutions and exploring data augmentation techniques more deeply, potentially offering substantial improvements.

In the next section, some possible ways to improve the model are proposed.

6.1 Further improvements and discussion

6.1.1 Auto-encoder

One potential improvement involves incorporating an auto-encoder [62]. An auto-encoder is a type of artificial neural network utilized in unsupervised learning, particularly within the domain of deep learning. Its primary purpose is to acquire efficient representations of input data by typically reducing its dimensionality. This architecture comprises two main components: an encoder and a decoder. The encoder takes the input data and transforms it into a lower-dimensional representation known as a bottleneck layer. Subsequently, the decoder reconstructs the initial input data based on this reduced-dimensional representation. The underlying objective of an auto-encoder is to learn how to capture the most crucial attributes of the input data while disregarding unnecessary or noisy details.
The intention here would be to generate fresh input features capable of expressing various cell populations. By reducing the dimensionality of markers, we could obtain higher-level features exclusively. These features could then be introduced as input to the fully connected part of our network (the convolutional layers are replaced by high level features input), contributing to the reduction of its complexity. This approach holds the potential to enhance the model ability to extract pertinent information from the data and could pave the way for more accurate predictions. However, efficiently summarizing the inputs by a limited number of high level features might turn out to be a task that is as difficult, or even more difficult, than the initial prediction task.

6.1.2 Histogram CNN

Another potential option for further improvement involves the concept of a histogram-based CNN. This approach could potentially address the issue of slow convergence in our model. Throughout our experiments, we noticed that despite using a high number of epochs, there appeared to be untapped performance potential that might be unlocked with even more epochs. This phenomenon could be attributed to the gradient becoming diluted as it passes through the average pooling layer. This layer calculates the average of the second convolution layer's output across half of the total cell count. As a result, the gradients spread out, leading to relatively minor adjustments in the convolutional layers.
To circumvent this challenge, an alternative could be to forgo the averaging step and maintain a more comprehensive record of marker distributions. This approach would still retain a conventional convolutional methodology while also accommodating varying cell populations.
The idea involves generating paired histograms for each combination of the 16 markers. Think of it as constructing a scatterplot matrix or a 2D joint histogram matrix. By segmenting the data into 16 bins, a 256 by 256 image emerges, with varying shades of gray representing the heights of the bins. The smooth transition between these bins would allow us to apply standard 3x3 convolutional filters and 2x2 pooling, thus reverting to a more typical utilization of convolutional neural networks. As a result, the initial cell count ceases to be a concern, since the relevant information is preserved within the paired histograms and pooling process conducted prior.
When it comes to comparing stimulated and non-stimulated data, one approach could involve juxtaposing the two "population images," resulting in a 512 by 256 arrangement. Alternatively, a method akin to using two color channels in RGB images might be suitable. Furthermore, given the symmetry of the data, an interesting presentation could involve placing stimulated data in the upper triangle and unstimulated data in the lower triangle.
By adopting this data encoding methodology, the deep learning and CNN aspects of the model become more intuitive and manageable. This approach offers potential avenues for enhanced performance and sheds light on the broader possibilities within the realm of convolutional neural networks.

References

1. *HIV* en. https://www.who.int/data/gho/data/themes/hiv-aids (2023).
2. *HIV and AIDS Epidemic Global Statistics* en. https://www.hiv.gov/hiv-basics/overview/data-and-trends/global-statistics (2023).
3. Aghaeepour, N. *et al.* A benchmark for evaluation of algorithms for identification of cellular correlates of clinical outcomes. eng. *Cytometry. Part A: The Journal of the International Society for Analytical Cytology* **89,** 16–21. ISSN: 1552-4930 (Jan. 2016).
4. Weintrob, A. C. *et al.* Increasing age at HIV seroconversion from 18 to 40 years is associated with favorable virologic and immunologic responses to HAART. eng. *Journal of acquired immune deficiency syndromes (1999)* **49,** 40–47. ISSN: 1944-7884. https://doi.org/10.1097/QAI.0b013e31817bec05 (2023) (Sept. 2008).
5. Hu, Z., Tang, A., Singh, J., Bhattacharya, S. & Butte, A. J. A robust and interpretable end-to-end deep learning model for cytometry data. *Proceedings of the National Academy of Sciences* **117.** Publisher: Proceedings of the National Academy of Sciences, 21373–21380. https://www.pnas.org/doi/10.1073/pnas.2003026117 (2023) (Sept. 2020).
6. *What are Neural Networks? | IBM* en-us. https://www.ibm.com/topics/neural-networks (2023).
7. *Artificial neural network* en. Page Version ID: 1169294231. Aug. 2023. https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=1169294231 (2023).
8. Grossi, E. & Buscema, M. Introduction to artificial neural networks. *European journal of gastroenterology & hepatology* **19,** 1046–54 (Jan. 2008).
9. *Deep Learning Neural Networks Explained in Plain English* en. June 2020. https://www.freecodecamp.org/news/deep-learning-neural-networks-explained-in-plain-english/ (2023).
10. *What is a Neural Network?* en. https://www.tibco.com/reference-center/what-is-a-neural-network (2023).
11. Bushaev, V. *How do we 'train' neural networks ?* en. Oct. 2018. https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73 (2023).
12. Kumar, S. *Common Loss functions in machine learning for a Regression model* en. Aug. 2020. https://medium.com/analytics-vidhya/common-loss-functions-in-machine-learning-for-a-regression-model-27d2bbda9c93 (2023).

13. Godoy, D. *Understanding binary cross-entropy / log loss: a visual explanation* en. July 2022. https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a (2023).
14. *Convolutional Neural Networks - Basics · Machine Learning Notebook* https://mlnotebook.github.io/post/CNN1/ (2023).
15. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way | Saturn Cloud Blog* en. Section: blog. Dec. 2018. https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/ (2023).
16. Sultana, F., Sufian, A. & Dutta, P. *Advancements in Image Classification using Convolutional Neural Network* en. in *2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)* arXiv:1905.03288 [cs] (Nov. 2018), 122–129. http://arxiv.org/abs/1905.03288 (2023).
17. Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. R. *Improving neural networks by preventing co-adaptation of feature detectors* arXiv:1207.0580 [cs]. July 2012. http://arxiv.org/abs/1207.0580 (2023).
18. Brownlee, J. *A Gentle Introduction to Dropout for Regularizing Deep Neural Networks* en-US. Dec. 2018. https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/ (2023).
19. Ioffe, S. & Szegedy, C. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* arXiv:1502.03167 [cs]. Mar. 2015. http://arxiv.org/abs/1502.03167 (2023).
20. Gupta, L. *Batch Normalization and ReLU for solving Vanishing Gradients* en. Apr. 2021. https://medium.com/analytics-vidhya/how-batch-normalization-and-relu-solve-vanishing-gradients-3f1a8ace1c88 (2023).
21. *Vanishing gradient problem* en. Page Version ID: 1166539994. July 2023. https://en.wikipedia.org/w/index.php?title=Vanishing_gradient_problem&oldid=1166539994 (2023).
22. *What is survival analysis? Examples by hand and in R* en. https://statsandr.com/blog/what-is-survival-analysis/ (2023).
23. Clark, T. G., Bradburn, M. J., Love, S. B. & Altman, D. G. Survival Analysis Part I: Basic concepts and first analyses. *British Journal of Cancer* **89,** 232–238. ISSN: 0007-0920. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2394262/ (2023) (July 2003).
24. *The notion of Censoring in Survival Analysis | JiGSO* en-US. Section: Employee Turnover. Feb. 2021. https://jigso.com/the-notion-of-censoring-in-survival-analysis/ (2023).
25. RICH, J. T. *et al.* A PRACTICAL GUIDE TO UNDERSTANDING KAPLAN-MEIER CURVES. *Otolaryngology–head and neck surgery : official journal of American Academy of Otolaryngology-Head and Neck Surgery* **143,** 331–336. ISSN: 0194-5998. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3932959/ (2023) (Sept. 2010).

26. *Kaplan–Meier estimator* en. Page Version ID: 1163438906. July 2023. https://en.wikipedia.org/w/index.php?title=Kaplan%E2%80%93Meier_estimator&oldid=1163438906 (2023).
27. *Kaplan-Meier Survival Estimates (Survival Curves) - StatsDirect* https://www.statsdirect.com/help/survival_analysis/kaplan_meier.htm (2023).
28. Paemel, R. V. *Kaplan Meier curves* en. July 2019. https://towardsdatascience.com/kaplan-meier-curves-c5768e349479 (2023).
29. *Proportional hazards model* en. Page Version ID: 1165810139. July 2023. https://en.wikipedia.org/w/index.php?title=Proportional_hazards_model&oldid=1165810139 (2023).
30. *Cox Proportional-Hazards Model - Easy Guides - Wiki - STHDA* en. http://www.sthda.com/english/wiki/cox-proportional-hazards-model (2023).
31. Waagepetersen, R. Cox's proportional hazards model and Cox's partial likelihood. en.
32. Alabdallah, A., Ohlsson, M., Pashami, S. & Rognvaldsson, T. *The Concordance Index decomposition: a measure for a deeper understanding of survival prediction models* (Feb. 2022).
33. Longato, E., Vettoretti, M. & Di Camillo, B. A practical perspective on the concordance index for the evaluation and selection of prognostic time-to-event models. en. *Journal of Biomedical Informatics* **108,** 103496. ISSN: 1532-0464. https://www.sciencedirect.com/science/article/pii/S1532046420301246 (2023) (Aug. 2020).
34. Rowley, T. Flow Cytometry - A Survey and the Basics. en. *Materials and Methods.* https://www.labome.com/method/Flow-Cytometry-A-Survey-and-the-Basics.html (2023) (Feb. 2023).
35. *ISAC - What is Cytometry?* https://isac-net.org/page/What-is-Cytometry (2023).
36. *Flow cytometry* en. Page Version ID: 1169126778. Aug. 2023. https://en.wikipedia.org/w/index.php?title=Flow_cytometry&oldid=1169126778 (2023).
37. Taylor, I. *Before FlowJo™ | FlowJo, LLC* English. https://www.flowjo.com/learn/flowjo-university/flowjo/before-flowjo/58 (2023).
38. PhD, T. B. *5 Gating Strategies To Get Your Flow Cytometry Data Published In Peer-Reviewed Scientific Journals* en-US. Aug. 2016. https://expert.cheekyscientist.com/gating-strategies-get-your-flow-cytometry-data-published/ (2023).
39. Emmaneel, A. *et al.* PeacoQC: Peak-based selection of high quality cytometry data. eng. *Cytometry. Part A: The Journal of the International Society for Analytical Cytology* **101,** 325–338. ISSN: 1552-4930 (Apr. 2022).

40. *Why Biexponential Transformation?* en-US. https://docs.flowjo.com/flowjo/graphs-and-gating/gw-transform-overview/gw-transform-benefits/ (2023).
41. *Understanding Data Scaling - OMIQ* en-US. https://www.omiq.ai/guides/understanding-data-scaling (2023).
42. Zhu, X., Yao, J. & Huang, J. *Deep convolutional neural network for survival analysis with pathological images* in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (Dec. 2016), 544–547.
43. Li, H. *et al.* DEEP CONVOLUTIONAL NEURAL NETWORKS FOR IMAGING DATA BASED SURVIVAL ANALYSIS OF RECTAL CANCER. *Proceedings. IEEE International Symposium on Biomedical Imaging* **2019,** 846–849. ISSN: 1945-7928. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6955095/ (2023) (Apr. 2019).
44. Zhang, Y. *et al.* CNN-based survival model for pancreatic ductal adenocarcinoma in medical imaging. *BMC Medical Imaging* **20,** 11. ISSN: 1471-2342. https://doi.org/10.1186/s12880-020-0418-1 (2023) (Feb. 2020).
45. Berrar, D. in *Encyclopedia of Bioinformatics and Computational Biology* 542–545 (Elsevier, 2019). ISBN: 978-0-12-809633-8.
46. Deng, L. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Processing Magazine* **29.** Conference Name: IEEE Signal Processing Magazine, 141–142. ISSN: 1558-0792 (Nov. 2012).
47. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* **15,** 1929–1958. ISSN: 1532-4435 (Jan. 2014).
48. Shorten, C. & Khoshgoftaar, T. M. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data* **6,** 60. ISSN: 2196-1115. https://doi.org/10.1186/s40537-019-0197-0 (2023) (July 2019).
49. Van Gassen, S., Vens, C., Dhaene, T., Lambrecht, B. N. & Saeys, Y. FloReMi: Flow density survival regression using minimal feature redundancy. eng. *Cytometry. Part A: The Journal of the International Society for Analytical Cytology* **89,** 22–29. ISSN: 1552-4930 (Jan. 2016).
50. Uno, H., Cai, T., Pencina, M. J., D'Agostino, R. B. & Wei, L. J. On the C-statistics for evaluating overall adequacy of risk prediction procedures with censored survival data. en. *Statistics in Medicine* **30.** _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sim 1105–1117. ISSN: 1097-0258. https://onlinelibrary.wiley.com/doi/abs/10.1002/sim.4154 (2023) (2011).
51. Pölsterl, S. *Evaluating Survival Models* en-us. May 2019. https://k-d-w.org/blog/2019/05/evaluating-survival-models/ (2023).
52. *Cross-Validation* en. https://nl.mathworks.com/discovery/cross-validation.html (2023).
53. *Quantile* en. Page Version ID: 1160588695. June 2023. https://en.wikipedia.org/w/index.php?title=Quantile&oldid=1160588695 (2023).

54. Team, T. A. *Choosing a Learning Rate for DNNs – Towards AI* en-US. https://towardsai.net/p/l/choosing-a-learning-rate-for-dnns,https://towardsai.net/p/l/choosing-a-learning-rate-for-dnns (2023).

55. *Adam - Cornell University Computational Optimization Open Textbook - Optimization Wiki* https://optimization.cbe.cornell.edu/index.php?title=Adam (2023).

56. Lever, J., Krzywinski, M. & Altman, N. Model selection and overfitting. en. *Nature Methods* **13.** Number: 9 Publisher: Nature Publishing Group, 703–704. ISSN: 1548-7105. https://www.nature.com/articles/nmeth.3968 (2023) (Sept. 2016).

57. Ng, A. Y. *Feature selection, L1 vs. L2 regularization, and rotational invariance* in *Proceedings of the twenty-first international conference on Machine learning* (Association for Computing Machinery, New York, NY, USA, July 2004), 78. ISBN: 978-1-58113-838-2. https://doi.org/10.1145/1015330.1015435 (2023).

58. Tewari, U. *Regularization — Understanding L1 and L2 regularization for Deep Learning* en. Nov. 2021. https://medium.com/analytics-vidhya/regularization-understanding-l1-and-l2-regularization-for-deep-learning-a7b9e4a409bf (2023).

59. *A Complete Guide to Data Augmentation* en-US. https://www.datacamp.com/tutorial/complete-guide-data-augmentation (2023).

60. *68–95–99.7 rule* en. Page Version ID: 1169537745. Aug. 2023. https://en.wikipedia.org/w/index.php?title=68%E2%80%9395%E2%80%9399.7_rule&oldid=1169537745 (2023).

61. Watts, V. 7.4 Confidence Intervals for a Population Proportion. en-ca. Book Title: Introduction to Statistics Publisher: Fanshawe College Pressbooks. https://ecampusontario.pressbooks.pub/introstats/chapter/7-4-confidence-intervals-for-a-population-proportion/ (2023) (Sept. 2022).

62. *Autoencoder* en. Page Version ID: 1168980551. Aug. 2023. https://en.wikipedia.org/w/index.php?title=Autoencoder&oldid=1168980551 (2023).

63. Faraggi, D. & Simon, R. A neural network model for survival data. en. *Statistics in Medicine* **14.** _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sim.4780140108, 73–82. ISSN: 1097-0258. https://onlinelibrary.wiley.com/doi/abs/10.1002/sim.4780140108 (2023) (1995).

64. Pölsterl, S. *Survival Analysis for Deep Learning* en-us. July 2019. https://k-d-w.org/blog/2019/07/survival-analysis-for-deep-learning/ (2023).

65. Han, X., Zhang, Y. & Shao, Y. On comparing two correlated C indices with censored survival data. *Statistics in medicine* **36,** 4041–4049. ISSN: 0277-6715. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5909734/ (2023) (Nov. 2017).

66. Kvamme, H., Borgan, & Scheel, I. *Time-to-Event Prediction with Neural Networks and Cox Regression* arXiv:1907.00825 [cs, stat]. Sept. 2019. http://arxiv.org/abs/1907.00825 (2023).

67. HARRELL Jr., F. E., Lee, K. L. & Mark, D. B. Multivariable Prognostic Models: Issues in Developing Models, Evaluating Assumptions and Adequacy, and Measuring and Reducing Errors. en. *Statistics in Medicine* **15.** _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291097-0258%2819960229%2915%3A4%3C361%3A%3AAID-SIM168%3E3.0.CO%3B2-4, 361–387. ISSN: 1097-0258. https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-0258%2819960229%2915%3A4%3C361%3A%3AAID-SIM168%3E3.0.CO%3B2-4 (2023) (1996).

68. Nowlan, S. J. & Hinton, G. E. Simplifying Neural Networks by Soft Weight-Sharing. *Neural Computation* **4.** Conference Name: Neural Computation, 473–493. ISSN: 0899-7667 (July 1992).

69. Rodger, A. J. *et al.* Risk of HIV transmission through condomless sex in serodifferent gay couples with the HIV-positive partner taking suppressive antiretroviral therapy (PARTNER): final results of a multicentre, prospective, observational study. English. *The Lancet* **393.** Publisher: Elsevier, 2428–2438. ISSN: 0140-6736, 1474-547X. https://www.thelancet.com/journals/lancet/article/PIIS0140-6736(19)30418-0/fulltext (2023) (June 2019).

Appendix A

Code used for this work

1.1 Data retrieval code

```
import numpy as np
import pandas as pd
import os
import pickle
import random as rd
from random import choice

flowCap_files = pd.read_csv("MetaDataFull.csv")
flowCap_train = pd.read_csv("MetaDataTrain.csv")


def modifyCytofFiles(cytof_files, filename):
    # Function to modify the names of the
    # cytometry files, the name changed after preprocessing

    cytof_files["Stim"] = cytof_files["Stim"].astype("string")
    cytof_files["Unstim"] = cytof_files["Unstim"].astype("string")
    cytof_files["Stim"] = cytof_files["Stim"].str.replace(".fcs", "")
    cytof_files["Unstim"] = cytof_files["Unstim"].str.replace(".fcs", "")
    cytof_files["Stim"] = cytof_files["Stim"] + "_preprocessed.csv"
    cytof_files["Unstim"] = cytof_files["Unstim"] + "_preprocessed.csv"
    cytof_files.to_csv(filename)


def retieve_data(cytof_files, train=True, n_cells=50000, random_state=42):
    if train:
        fn = [
            [
                os.path.join(os.getcwd(), "FlowCap_preProcessed", stim),
                os.path.join(os.getcwd(), "FlowCap_preProcessed", unstim),
            ]
            for stim, unstim, status in zip(
                cytof_files.Stim, cytof_files.Unstim, cytof_files.Status
            )
            if not np.isnan(status)
        ]
```

```
        time_event_array = np.zeros(
            shape=(len(fn),),
            dtype=np.dtype([("Censored", np.uint16), ("Time", np.float16)]),
        )
        time_event_array["Censored"] = \
            np.array(flowCap_train["Status"].iloc[:191])
        time_event_array["Time"] = \
            np.array(flowCap_train["Survival Time"].iloc[:191])
    else:
        fn = [
            [
                os.path.join(os.getcwd(), "FlowCap_preProcessed", stim),
                os.path.join(os.getcwd(), "FlowCap_preProcessed", unstim),
            ]
            for stim, unstim, status, idx in zip(
                cytof_files.Stim,
                cytof_files.Unstim,
                cytof_files.Status,
                cytof_files.Index,
            )
            if not np.isnan(status) and idx > 190
        ]

        time_event_array = np.zeros(
            shape=(len(fn),),
            dtype=np.dtype([("Censored", np.uint16), ("Time", np.float16)]),
        )

        time_event_array["Censored"] = \
            np.array(flowCap_files["Status"])[191:]

        time_event_array["Time"] = \
            np.array(flowCap_files["Survival Time"])[191:]

    data = np.zeros(shape=(len(fn), n_cells, 16, 1), dtype=np.float16)
    for i in range(len(fn)):
        [fs, fu] = fn[i]
        stim = pd.read_csv(fs)
        unstim = pd.read_csv(fu)
        stim = stim.drop(columns=["Time", "Original_ID"])
        unstim = unstim.drop(columns=["Time", "Original_ID"])
        stim = np.asarray(
            stim.sample(
                n=n_cells // 2,
                replace=True,
                random_state=random_state,
                ignore_index=True,
            )
        ).reshape(n_cells, 16, 1)

        unstim = np.asarray(
            unstim.sample(
                n=n_cells // 2,
                replace=True,
                random_state=random_state,
```

```
                ignore_index=True,
            )
        ).reshape(n_cells, 16, 1)

        np.concatenate([stim, unstim], axis=0, out=data[i])
    return data, time_event_array


def data_augm(time_event_array, data, n):
    # Data augmentation function, n length of the final data
    l = data.shape[0]
    n_cells = data.shape[1]
    i = l
    data_aug = np.zeros(shape=(n, n_cells, 16, 1), dtype=np.float16)
    time_event_array_augm = np.zeros(
        shape=(n,), dtype=np.dtype([("Censored", np.uint16),
                                    ("Time", np.float16)])
    )
    data_aug[:i] = data
    time_event_array_augm[:i] = time_event_array
    while i < n:
        rand = rd.randint(0, l - 1)
        noise = np.random.normal(0, 0.5, 1)

        time_event_array_augm["Time"][i] = \
            time_event_array["Time"][rand] + noise

        time_event_array_augm["Censored"][i] = \
            time_event_array["Censored"][rand]

        data_aug[i] = data[rand]
        i += 1
    return data_aug, time_event_array_augm


def add_to_split(i, qt, counter, cnt_idx, time_array, splits):
    cnt = counter[cnt_idx]
    maxi = len(np.where(time_array < np.quantile(time_array, qt))[0]) \
        - len(np.where(time_array < np.quantile(time_array, qt - 0.1))[0])

    pos = [idx for idx, val in enumerate(cnt) if val < int(maxi // 4)]
    if len(pos) == 0:
        pos = [i for i in range(4) if cnt[i] < min(cnt) + 1]
    rd = choice(pos)
    splits[rd].append(i)
    counter[cnt_idx, rd] += 1


def check_quantile(time, time_array, n_quantiles):
    for i in range(n_quantiles):
        qt = i / 10 + 0.1
        if time < np.quantile(time_array, qt):
            break
    return qt, i
```

```
def create_splits(time_event_array, n_splits, augmented=True):
    if augmented:
        event_real = time_event_array["Censored"].astype(np.int16)[:191]
        time_real = time_event_array["Time"].astype(np.float16)[:191]
        time_event_real = event_real * time_real
        time_not_censored_real = time_event_real[np.where(time_event != 0)]
        inv_time_event_real = (1 - event_real) * time_real
        time_censored_real = inv_time_event_real\
            [np.where(inv_time_event_real != 0)]

        pos_time_censored_real = np.where(inv_time_event_real != 0)
        event_augm = time_event_array["Censored"].astype(np.int16)[191:]
        time_augm = time_event_array["Time"].astype(np.float16)[191:]
        time_event_augm = event_augm * time_augm
        time_not_censored_augm = time_event_augm[np.where(time_event != 0)]
        inv_time_event_augm = (1 - event_augm) * time_augm
        time_censored_augm = inv_time_event_augm\
            [np.where(inv_time_event_augm != 0)]

        pos_time_censored_augm = np.where(inv_time_event_augm != 0)
    else:
        event = time_event_array["Censored"].astype(np.int16)
        time = time_event_array["Time"].astype(np.float16)
        time_event = event * time
        time_not_censored = time_event[np.where(time_event != 0)]
        inv_time_event = (1 - event) * time
        time_censored = inv_time_event[np.where(inv_time_event != 0)]
        pos_time_censored = np.where(inv_time_event != 0)

    splits = [[] for i in range(n_splits)]

    if augmented:
        counter = np.zeros(shape=(20, 4))
        n_quantiles = 10
        for i in range(len(time_real)):
            if i in pos_time_censored_real[0]:
                qt, cnt_idx = check_quantile(
                    time_real[i], time_censored_real, n_quantiles
                )
                create_splits(i, qt, counter,
                              cnt_idx, time_censored_real, splits)
            else:
                qt, cnt_idx = check_quantile(
                    time_real[i], time_not_censored_real, n_quantiles
                )
                create_splits(
                    i, qt, counter,
                    cnt_idx + 10, time_not_censored_real, splits
                )

        counter = np.zeros(shape=(20, 4))
        for i in range(len(time_augm)):
            if i in pos_time_censored_augm[0]:
                qt, cnt_idx = check_quantile(
```

```
                    time_augm[i], time_censored_augm, n_quantiles
                )
                create_splits(i, qt, counter,
                              cnt_idx, time_censored_augm, splits)
            else:
                qt, cnt_idx = check_quantile(
                    time_augm[i], time_not_censored_augm, n_quantiles
                )
                create_splits(
                    i, qt, counter,
                    cnt_idx + 10, time_not_censored_augm, splits
                )

    else:
        counter = np.zeros(shape=(20, 4))
        n_quantiles = 10
        for i in range(len(time)):
            if i in pos_time_censored[0]:
                qt, cnt_idx = \
                    check_quantile(time[i], time_censored, n_quantiles)

                create_splits(i, qt, counter,
                              cnt_idx, time_censored, splits)
            else:
                qt, cnt_idx = \
                    check_quantile(time[i], time_not_censored, n_quantiles)

                create_splits(i, qt, counter,
                              cnt_idx + 10, time_not_censored, splits)

    splits = [np.array(sublist) for sublist in splits]
    splits = np.array(splits, dtype=object)

    return splits


def save_file(filename, data):
    with open(filename, "wb") as f:
        pickle.dump(data, f)


if __name__ == "__main__":
    # modifyCytofFiles(flowCap_train, "MetaDataTrain.csv")
    data, time_event_array = retieve_data(
        flowCap_train, train=True, n_cells=20000, random_state=42
    )
    data, time_event_array = data_augm(time_event_array, data, n=1000)
    splits = create_splits(time_event_array, n_splits=4, augmented=True)
    save_file("data_10000_20000.pkl", data)
    save_file("time_event_10000_20000.pkl", time_event_array)
    save_file("splits_10000_20000.pkl", splits)
```

1.2 Model training and testing code

```
import keras
from keras.layers import Dense, Flatten, BatchNormalization, Activation, \
    Conv2D, AveragePooling2D, Input, Dropout
from keras.models import Model
import pickle
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf;
from typing import Dict, Optional, Sequence
from sksurv.metrics import concordance_index_ipcw


def load_data(filename):
    file = open(filename, "rb")
    data = pickle.load(file)
    return data


def _make_riskset(time: np.ndarray) -> np.ndarray:
    """Compute mask that represents each sample's risk set.

    Parameters
    ----------
    time : np.ndarray, shape=(n_samples,)
        Observed event time sorted in descending order.

    Returns
    -------
    risk_set : np.ndarray, shape=(n_samples, n_samples)
        Boolean matrix where the `i`-th row denotes the
        risk set of the `i`-th instance, i.e. the indices `j`
        for which the observer time `y_j >= y_i`.
    """
    assert time.ndim == 1, "expected 1D array"

    # sort in descending order
    o = np.argsort(-time, kind="mergesort")
    n_samples = len(time)
    risk_set = np.zeros((n_samples, n_samples), dtype=np.bool_)
    for i_org, i_sort in enumerate(o):
        ti = time[i_sort]
        k = i_org
        while k < n_samples and ti == time[o[k]]:
            k += 1
        risk_set[i_sort, o[:k]] = True
    return risk_set


def safe_normalize(x: tf.Tensor) -> tf.Tensor:
    """Normalize risk scores to avoid exp underflowing.

    Note that only risk scores relative to each other matter.
    If minimum risk score is negative, we shift scores so minimum
```

```
    is at zero.
    """
    x_min = tf.reduce_min(x, axis=0)
    c = tf.zeros_like(x_min)
    norm = tf.where(x_min < 0, -x_min, c)
    return x + norm


def logsumexp_masked(risk_scores: tf.Tensor,
                     mask: tf.Tensor,
                     axis: int = 0,
                     keepdims: Optional[bool] = None) -> tf.Tensor:
    """Compute logsumexp across `axis` for entries where `mask` is true."""
    risk_scores.shape.assert_same_rank(mask.shape)

    with tf.name_scope("logsumexp_masked"):
        mask_f = tf.cast(mask, risk_scores.dtype)
        risk_scores_masked = tf.math.multiply(risk_scores, mask_f)
        # for numerical stability, substract the maximum value
        # before taking the exponential
        amax = tf.reduce_max(risk_scores_masked, axis=axis, keepdims=True)
        risk_scores_shift = risk_scores_masked - amax

        exp_masked = tf.math.multiply(tf.exp(risk_scores_shift), mask_f)
        exp_sum = tf.reduce_sum(exp_masked, axis=axis, keepdims=True)
        output = amax + tf.math.log(exp_sum)
        if not keepdims:
            output = tf.squeeze(output, axis=axis)
    return output


class CoxPHLoss(tf.keras.losses.Loss):
    """Negative partial log-likelihood of Cox's proportional hazards model."""

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

    def call(self,
             y_true: Sequence[tf.Tensor],
             y_pred: tf.Tensor) -> tf.Tensor:
        """Compute loss.

        Parameters
        ----------
        y_true : list|tuple of tf.Tensor
            The first element holds a binary vector where 1
            indicates an event 0 censoring.
            The second element holds the riskset, a
            boolean matrix where the `i`-th row denotes the
            risk set of the `i`-th instance, i.e. the indices `j`
            for which the observer time `y_j >= y_i`.
            Both must be rank 2 tensors.
        y_pred : tf.Tensor
            The predicted outputs. Must be a rank 2 tensor.
```

```
    Returns
    -------
    loss : tf.Tensor
        Loss for each instance in the batch.
    """
    event, riskset = y_true
    event = tf.convert_to_tensor(event)
    riskset = tf.convert_to_tensor(riskset)
    predictions = y_pred

    pred_shape = predictions.shape
    if pred_shape.ndims != 2:
        raise ValueError("Rank mismatch: Rank of predictions \
                         (received %s) should "
                         "be 2." % pred_shape.ndims)

    if pred_shape[1] is None:
        raise ValueError("Last dimension of predictions must be known.")

    if pred_shape[1] != 1:
        raise ValueError("Dimension mismatch: \
                         Last dimension of predictions "
                         "(received %s) must be 1." % pred_shape[1])

    if event.shape.ndims != pred_shape.ndims:
        raise ValueError("Rank mismatch: Rank of predictions \
                         (received %s) should "
                         "equal rank of event (received %s)" % (
            pred_shape.ndims, event.shape.ndims))

    if riskset.shape.ndims != 2:
        raise ValueError("Rank mismatch: Rank of riskset \
                         (received %s) should "
                         "be 2." % riskset.shape.ndims)

    event = tf.cast(event, predictions.dtype)
    predictions = safe_normalize(predictions)


    with tf.name_scope("assertions"):
        assertions = (
            tf.debugging.assert_less_equal(event, 1.),
            tf.debugging.assert_greater_equal(event, 0.),
            tf.debugging.assert_type(riskset, tf.bool)
        )

    # move batch dimension to the end so predictions get broadcast
    # row-wise when multiplying by riskset
    pred_t = tf.transpose(predictions)
    # compute log of sum over risk set for each row
    rr = logsumexp_masked(pred_t, riskset, axis=1, keepdims=True)
    assert rr.shape.as_list() == predictions.shape.as_list()

    losses = tf.math.multiply(event, rr - predictions)
    return losses
```

```
class CindexMetric:
    """Computes concordance index across one epoch."""

    def reset_states(self) -> None:
        """Clear the buffer of collected values."""
        self._data = {
            "train": [],
            "test": [],
            "prediction": []
        }

    def update_state(self, y_train: Dict[str, tf.Tensor],
                     y_test: Dict[str, tf.Tensor], y_pred: tf.Tensor) -> None:


        self._data["train"] = np.zeros(shape=(len(y_train["label_event"]),),
                                       dtype=np.dtype([('event', bool), ('time', float)]

        self._data["test"] = np.zeros(shape=(len(y_test["label_event"]),),
                                      dtype=np.dtype([('event', bool), ('time', float)])

        self._data["prediction"] = tf.squeeze(y_pred).numpy()
        self._data["train"]["event"] = \
            tf.cast(y_train["label_event"], tf.bool).numpy()

        self._data["train"]["time"] = y_train["label_time"]
        self._data["test"]["event"] = \
            tf.cast(y_test["label_event"], tf.bool).numpy()

        self._data["test"]["time"] = y_test["label_time"]


    def result(self) -> Dict[str, float]:
        """Computes the concordance index across collected values."""

        results = concordance_index_ipcw(
            self._data["train"],
            self._data["test"],
            self._data["prediction"])

        result_data = {}
        names = ("cindex", "concordant", "discordant", "tied_risk")
        for k, v in zip(names, results):
            result_data[k] = v

        return result_data


class TrainAndEvaluateModel:

    def __init__(self, model_func, model_dir, data, time, event,
                 splits, learning_rate, num_epochs,
                 size_conv_1, size_conv_2, size_dense_1,
```

```
                size_dense_2, size_output, lmbda ):

        self.num_epochs = num_epochs
        self.dataset = data
        self.time = time
        self.event = event
        self.n_fold = len(splits)
        self.models = list()
        self.optimizers = list()
        self.splits = list()
        for i in range(self.n_fold):
          left_splits = [splits[k] for k in range(self.n_fold) if k != i]
          self.splits.append((np.concatenate(left_splits, axis=None), splits[i]))

        for _ in range(self.n_fold):
          self.models.append(model_func(data, size_conv_1,
                                        size_conv_2, size_dense_1,
                                        size_dense_2, size_output, lmbda))
          self.optimizers.append(\
              tf.keras.optimizers.Adam(learning_rate=learning_rate))


        self.loss_fn = CoxPHLoss()
        self.train_loss = np.zeros(shape=(self.num_epochs, self.n_fold))
        self.val_loss = np.zeros(shape=(self.num_epochs, self.n_fold))
        self.train_cindex = np.zeros(shape=(self.num_epochs, self.n_fold))
        self.val_cindex = np.zeros(shape=(self.num_epochs, self.n_fold))
        self.init_cindex = np.zeros(shape=(self.num_epochs, self.n_fold))

        self.val_cindex_metric = CindexMetric()
        self.train_cindex_metric = CindexMetric()

    def save_file(self, filename, data):
        with open(filename,'wb') as f:
            pickle.dump(data, f)

    def save(self):
        i=0
        for model in self.models:
            model.save("model_fold_" + str(i)+".h5")
            i += 1
        self.save_file("train_loss.pkl" ,self.train_loss)
        self.save_file("val_loss.pkl" ,self.val_loss)
        self.save_file("train_cindex.pkl" ,self.train_cindex)
        self.save_file("val_cindex.pkl" ,self.val_cindex)
        self.save_file("init_cindex.pkl", self.init_cindex)

    def plots(self):

        mean_train_loss = np.mean(self.train_loss, axis=1)
        mean_valid_loss = np.mean(self.val_loss, axis=1)
        mean_train_cindex = np.mean(self.train_cindex, axis=1)
        mean_valid_cindex = np.mean(self.val_cindex, axis=1)
        mean_init_cindex = np.mean(self.init_cindex, axis=1)
```

```
        # plt.figure()
        # labels=["Fold"+str(i) for i in range(self.train_loss.T.shape[0])]
        # for y_arr, label in zip(self.train_loss.T, labels):
        #     plt.plot([i for i in range(1, self.num_epochs+1)],
        #              y_arr, label=label)
        # plt.title("Train loss")
        # plt.xlabel("Epoch")
        # plt.ylabel("Train loss")
        # plt.legend()
        # plt.show()

        # plt.figure()
        # for y_arr, label in zip(self.train_cindex.T, labels):
        #     plt.plot([i for i in range(1, self.num_epochs+1)],
        #              y_arr, label=label)
        # plt.title("Train cindex")
        # plt.xlabel("Epoch")
        # plt.ylabel("Train cindex")
        # plt.legend()
        # plt.show()

        # plt.figure()
        # for y_arr, label in zip(self.val_loss.T, labels):
        #     plt.plot([i for i in range(1, self.num_epochs+1)],
        #              y_arr, label=label)
        # plt.title("Validation loss")
        # plt.xlabel("Epoch")
        # plt.ylabel("Validation loss")
        # plt.legend()
        # plt.show()

        # plt.figure()
        # for y_arr, label in zip(self.val_cindex.T, labels):
        #     plt.plot([i for i in range(1, self.num_epochs+1)],
        #              y_arr, label=label)
        # plt.title("Validation cindex")
        # plt.xlabel("Epoch")
        # plt.ylabel("Validation cindex")
        # plt.legend()
        # plt.show()

        plt.figure()
        labels=["Mean Train Loss", "Mean Valid Loss"]
        for y_arr, label in zip([mean_train_loss, mean_valid_loss], labels):
            plt.plot([i for i in range(1, self.num_epochs+1)],
                     y_arr, label=label)
        plt.title("Mean loss")
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.legend()
        plt.show()

        plt.figure()
        labels=["Mean Train Cindex",
```

```
            "Mean Valid Cindex",
            "Mean Real Data Cindex"]

    for y_arr, label in zip([mean_train_cindex, mean_valid_cindex,
                             mean_init_cindex], labels):

        plt.plot([i for i in range(1, self.num_epochs+1)],
                 y_arr, label=label)
    plt.title("Mean cindex")
    plt.xlabel("Epoch")
    plt.ylabel("Cindex")
    plt.legend()
    plt.show()



def train_one_step(self, x, y_event, y_riskset, fold):
    y_event = tf.expand_dims(y_event, axis=1)
    with tf.GradientTape() as tape:
        logits = self.models[fold](x, training=True)
        train_loss = self.loss_fn(y_true=[y_event, y_riskset],
                                  y_pred=logits)
    grads = tape.gradient(train_loss,
                          self.models[fold].trainable_variables)

    self.optimizers[fold].apply_gradients(\
        zip(grads, self.models[fold].trainable_variables))

    tape.__exit__
    return train_loss/tf.cast(tf.reduce_sum(y_event), tf.float32), \
        logits

def train_and_evaluate(self):

    for epoch in range(self.num_epochs):
        print("Epoch {}".format(epoch+1))
        for fold, (train_index, valid_index) in enumerate(self.splits):
            train_data = self.dataset[train_index]
            train_label = {
                "label_event": self.event[train_index],
                "label_time": self.time[train_index],
                "label_riskset": _make_riskset(self.time[train_index])
                }
            self.train_one_fold(epoch, fold, train_data, train_label)

            # Run a validation loop at the end of each epoch.
            valid_data = self.dataset[valid_index]
            valid_label = {
                "label_event": self.event[valid_index],
                "label_time": self.time[valid_index],
                "label_riskset": _make_riskset(self.time[valid_index])
                }
            initial_index = [idx for idx in valid_index if idx<191]
            initial_data = self.dataset[initial_index]
            initial_label = {
```

```
                    "label_event": self.event[initial_index],
                    "label_time": self.time[initial_index],
                    "label_riskset": _make_riskset(self.time[initial_index])
                    }
                self.evaluate(epoch, fold, train_label, valid_data,
                              valid_label, initial_data, initial_label)

            if epoch == 0 or (epoch+1)%50 == 0:
              mean_train_loss = np.mean(self.train_loss[epoch,:])
              mean_valid_loss = np.mean(self.val_loss[epoch,:])
              mean_train_cindex = np.mean(self.train_cindex[epoch,:])
              mean_valid_cindex = np.mean(self.val_cindex[epoch,:])
              print("\n")
              print("Summary")
              print(f"Mean Train Kfold loss = {mean_train_loss:.6f}, \
                    mean cindex = {mean_train_cindex:.4f}")

              print(f"Mean Validation Kfold loss = {mean_valid_loss:.6f}, \
                    mean cindex = {mean_valid_cindex:.4f}")

              print("\n")
              print("\n")

    def train_one_fold(self, epoch, fold, x, y):
        self.train_cindex_metric.reset_states()

        train_loss, logits = self.train_one_step(
                x, y["label_event"], y["label_riskset"], fold, epoch)

        self.train_loss[epoch, fold] = train_loss
        self.train_cindex_metric.update_state(y, y, logits)
        train_cindex = self.train_cindex_metric.result()
        self.train_cindex[epoch, fold] = train_cindex['cindex']

        if epoch == 0 or (epoch+1)%50 == 0:
          print("Fold {}".format(fold+1))
          print(f"Train: loss = {train_loss:.6f}, \
                cindex = {train_cindex['cindex']:.4f}")

    @tf.function
    def evaluate_one_fold(self, fold, x, y_event, y_riskset, x_init):
        y_event = tf.expand_dims(y_event, axis=1)
        val_logits = self.models[fold](x, training=False)
        init_logits = self.models[fold](x_init, training=False)
        event = tf.concat([y_event, y_event, y_event], axis=0)
        riskset = tf.concat([y_riskset, y_riskset, y_riskset], axis=1)
        riskset = tf.concat([riskset, riskset, riskset], axis=0)
        logits = tf.concat([val_logits, val_logits, val_logits], axis=0)
        val_loss = self.loss_fn(y_true=[event, riskset], y_pred=logits)

        return val_loss/tf.cast(tf.reduce_sum(event), tf.float32), \
            val_logits, init_logits

    def evaluate(self, epoch, fold, y, x_val, y_val, x_init, y_init):
        self.val_cindex_metric.reset_states()
```

```
        val_loss, val_logits, init_logits = \
            self.evaluate_one_fold(fold, x_val,  y_val["label_event"],
                                   y_val["label_riskset"], x_init)

        # Update val metrics
        self.val_loss[epoch, fold] = val_loss
        self.val_cindex_metric.update_state(y, y_val, val_logits)
        val_cindex = self.val_cindex_metric.result()
        self.val_cindex_metric.reset_states()
        self.val_cindex_metric.update_state(y, y_init, init_logits)
        init_cindex = self.val_cindex_metric.result()
        self.val_cindex[epoch, fold] = val_cindex['cindex']
        self.init_cindex[epoch, fold] = init_cindex['cindex']
        if epoch == 0 or (epoch+1)%50 == 0:

          print(f"Validation: loss = {val_loss:.6f}, \
                cindex = {val_cindex['cindex']:.4f}")

          print(f"Initial Study Data: cindex = {init_cindex['cindex']:.4f}")
```

```
class TestModel:

    def __init__(self, model_func, data, time, event,
                 data_test, time_test, event_test,
                 learning_rate, num_epochs,
                 size_conv_1, size_conv_2, size_dense_1,
                 size_dense_2, size_output, lmbda ):

        self.num_epochs = num_epochs
        self.model = model_func(data, size_conv_1, size_conv_2,
                                size_dense_1, size_dense_2,
                                size_output, lmbda)

        self.optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
        self.train_data = data
        self.train_label = {
                             "label_event": event,
                             "label_time": time,
                             "label_riskset": _make_riskset(time)
                            }

        # Run a test loop at the end of each epoch.
        self.test_data = data_test
        self.test_label = {
                       "label_event": event_test,
                       "label_time": time_test,
                       "label_riskset": _make_riskset(time_test)
```

```
                    }

        self.loss_fn = CoxPHLoss()
        self.train_loss = np.zeros(shape=(self.num_epochs,))
        self.test_loss = np.zeros(shape=(self.num_epochs,))
        self.train_cindex = np.zeros(shape=(self.num_epochs,))
        self.test_cindex = np.zeros(shape=(self.num_epochs,))

        self.test_cindex_metric = CindexMetric()
        self.train_cindex_metric = CindexMetric()


    def save_file(self, filename, data):
        with open(filename,'wb') as f:
            pickle.dump(data, f)

    def save(self):

        self.model.save("model_test.h5")
        self.save_file("train_loss.pkl" ,self.train_loss)
        self.save_file("test_loss.pkl" ,self.test_loss)
        self.save_file("train_cindex.pkl" ,self.train_cindex)
        self.save_file("test_cindex.pkl" ,self.test_cindex)


    def plots(self):

        plt.figure()
        plt.plot([i for i in range(1, self.num_epochs+1)],
                 self.train_loss.T, label = "Train")

        plt.plot([i for i in range(1, self.num_epochs+1)],
                 self.test_loss.T, label= "Test")

        plt.title("Loss")
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.legend()
        plt.show()

        plt.figure()
        plt.plot([i for i in range(1, self.num_epochs+1)],
                 self.train_cindex.T, label = "Train")

        plt.plot([i for i in range(1, self.num_epochs+1)],
                 self.test_cindex.T, label= "Test")

        plt.title("Cindex")
        plt.xlabel("Epoch")
        plt.ylabel("Cindex")
        plt.legend()
        plt.show()

    def train_one_step(self):
```

```
        x = self.train_data
        y_riskset = self.train_label["label_riskset"]
        y_event = tf.expand_dims(self.train_label["label_event"], axis=1)

        with tf.GradientTape() as tape:
            logits = self.model(x, training=True)
            train_loss = self.loss_fn(y_true=[y_event, y_riskset],
                                      y_pred=logits)

        grads = tape.gradient(train_loss, self.model.trainable_variables)
        self.optimizer.apply_gradients(zip(grads,
                                           self.model.trainable_variables))

        return train_loss/tf.cast(tf.reduce_sum(y_event), tf.float32), logits

    def train_and_evaluate(self):

        for epoch in range(self.num_epochs):
            print("Epoch {}".format(epoch+1))
            self.train_one_epoch(epoch)
            self.evaluate(epoch)

    def train_one_epoch(self, epoch):
        self.train_cindex_metric.reset_states()

        train_loss, logits = self.train_one_step(epoch)

        self.train_loss[epoch] = train_loss
        self.train_cindex_metric.update_state(self.train_label,
                                              self.train_label, logits)
        train_cindex = self.train_cindex_metric.result()
        self.train_cindex[epoch] = train_cindex['cindex']

        if epoch < 10 or (epoch+1)%50 == 0:
          print(f"Train: loss = {train_loss:.6f}, \
                cindex = {train_cindex['cindex']:.4f}")

    @tf.function
    def evaluate_one_epoch(self):

        x = self.test_data
        y_riskset = self.test_label["label_riskset"]
        y_event = tf.expand_dims(self.test_label["label_event"], axis=1)
        test_logits = self.model(x, training=False)
        event = tf.concat([y_event, y_event, y_event,
                           y_event, y_event], axis=0)

        riskset = tf.concat([y_riskset, y_riskset,
                             y_riskset, y_riskset, y_riskset], axis=1)

        riskset = tf.concat([riskset, riskset,
                             riskset, riskset, riskset], axis=0)

        logits = tf.concat([test_logits, test_logits,
                            test_logits, test_logits, test_logits], axis=0)
```

```
        test_loss = self.loss_fn(y_true=[event, riskset], y_pred=logits)

        return test_loss/tf.cast(tf.reduce_sum(event), tf.float32), \
            test_logits

    def evaluate(self, epoch):
        self.test_cindex_metric.reset_states()

        test_loss, test_logits = self.evaluate_one_epoch()
        # Update test metrics
        self.test_loss[epoch] = test_loss
        self.test_cindex_metric.update_state(self.train_label,
                                             self.test_label, test_logits)
        test_cindex = self.test_cindex_metric.result()
        self.test_cindex[epoch] = test_cindex['cindex']
        if epoch < 10 or (epoch+1)%50 == 0:
          print(f"Test: loss = {test_loss:.6f}, \
                cindex = {test_cindex['cindex']:.4f}")

def create_CNN(input, size_conv_1, size_conv_2, lmbda):

    #First Convolutional Layer
    model_output = Conv2D(size_conv_1, kernel_size=(1, input.shape[2]),
                          activation="relu",
                          # kernel_regularizer=keras.regularizers.l2(lmbda),
                          name="Conv_1")(input)
    model_output = BatchNormalization()(model_output)
    model_output = Dropout(0.5)(model_output)


    #Second Convolutional Layer
    model_output = Conv2D(size_conv_2, kernel_size=(1, 1),
                   activation="relu",
                  #  kernel_regularizer=keras.regularizers.l2(lmbda),
                  name="Conv_2")(model_output)
    model_output = BatchNormalization()(model_output)
    model_output = Dropout(0.5)(model_output)


    #Pooling layer
    model_output = AveragePooling2D(pool_size=(input.shape[1]/2, 1),
                                    strides=(input.shape[1]/2, 1),
                                    name="Avg_Pooling")(model_output)

    model_output = Flatten(name="Flatten")(model_output)

    return model_output


def create_model(data, size_conv_1, size_conv_2,
                 size_dense_1, size_dense_2,
                 size_output, lmbda):

  input = Input(shape=data[0].shape, name="Input")
```

```
    input = Dropout(0.2)(input)
    cnn = create_CNN(input, size_conv_1, size_conv_2, lmbda)


    output = Dense(size_dense_1,
                   activation="relu",
                  #  kernel_regularizer=keras.regularizers.l2(lmbda),
                   name="Dense_1")(cnn)
    output = BatchNormalization()(output)
    output = Dropout(0.5)(output)



    output = Dense(size_dense_2,
                   activation="relu",
                  #  kernel_regularizer=keras.regularizers.l2(lmbda),
                   name="Dense_2")(output)
    output = BatchNormalization()(output)
    output = Dropout(0.5)(output)



    output = Dense(size_output,
                   activation="linear",
                  #  kernel_regularizer=keras.regularizers.l2(lmbda),
                   name="Output")(output)

    model = Model(inputs=[input], outputs=[output])

    return model



if __name__ == '__main__':
    data = load_data("data_1000_20000.pkl")
    data_test = load_data("data_test.pkl")
    time_event_array = load_data("time_event_10000_20000.pkl")
    time_event_array_test = load_data("time_event_test.pkl")
    # splits = load_data("splits_10000_20000.pkl")
    event = time_event_array["Censored"]
    time = time_event_array["Time"]
    event_test = time_event_array_test["Censored"]
    time_test = time_event_array_test["Time"]

    # trainer = TrainAndEvaluateModel(
    #     model_func=create_model,
    #     data=data,
    #     time=time,
    #     event=event,
    #     splits = splits,
    #     learning_rate=0.001,
    #     num_epochs=1000,
    #     size_conv_1=6,
    #     size_conv_2=6,
    #     size_dense_1=6,
```

```
#     size_dense_2=3,
#     size_output=1,
#     lmbda=None
# )

trainer = TestModel(
    model_func=create_model,
    data=data,
    time=time,
    event=event,
    data_test = data_test,
    time_test = time_test,
    event_test = event_test,
    learning_rate=0.001,
    num_epochs=1500,
    size_conv_1=6,
    size_conv_2=6,
    size_dense_1=6,
    size_dense_2=3,
    size_output=1,
    lmbda=None
)

trainer.save()
trainer.plots()
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**