Submitted in fulfillment of the requirements for the degree of
Master of Science in Computer Science:
Data Science

Predicting MHC binding preferences using recurrent neural networks

Bruno De Deken

Promotor: Prof. Dr. Kris Laukens

Co-Promotor: Dr. Pieter Meysman

Faculty of Science
2017

Contents

List of Abbreviations

ANN Artificial Neural Network

CNN Convolutional Neural Network

CTL Cytotoxic T-Lymphocyte or Cytotoxic T-Cell

DN Deep and Narrow

HLA Human Leukocyte Antigen

MHC Major Histocompatibility Complex

NK cells Natural Killer Cells

RNN Recurrent Neural Network

SM Simple Model

TCR T-Cell Receptor

WS Wide and Shallow

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis we will propose a new technique to predict the binding preferences of Major Histocompatibility Complex (MHC) antigens with peptides of various sequences. We perform these predictions with a deep learning model using a recurrent neural network. All data used during the training of this network is publicly available.

1.1 Project context

In immunological research, biologists examine the effects of active ingredients on cells taken from the blood of patients. These experiments are very time consuming and expensive but are also essential if we want to understand more about MHC antigen presentation. Research in MHC antigen presentation is a crucial component in cancer research, finding matching donor organs or developing new vaccins.
If researchers would combine samples of ingredients with arbitrarily chosen blood samples of patients, a lot of time and resources would be wasted since the chances of success are so small. This creates a very big limitation on medical and scientific progress. To solve this issue, researchers have

developed different analyzation tools that can suggest which experiments might yield interesting results. An added difficulty is that, for some of the relevant processes that happen between cells, very little data is available. We believe that a single solution can exist that could solve these problems: Creating a generic predictive tool. We hope to create a tool that can be trained on well known data in order to make useful predictions concerning this data, while at the same time be generic enough so that it can be used to make predictions on lesser known data.

Similar work Several tools exist that offer similar functionality, many of these tools are built upon already existing tools. Table 1.1 gives an overview of some popular tools with the underlying technologies used.

Tool	Main technology
NetMHCPan[1]	artificial neural network
PickPocket[2]	position specific scoring matrix
KISS[3]	Kernel
ADT[4]	Adaptive double threading

Table 1.1: *Overview of tools that offer similar functionality with the corresponding technology used*

1.2 Problem domain

MHC molecules capture a snapshot of the contents of the cell and present it on the cells surface. This enables specific immunological cells, called T-cells, to screen for foreign compounds to be eliminated. This is the foundation of any immune reaction. Succesfully detecting foreign cells and destroying them is what allows us to recover from illnesses caused by virusses or bacteria, while cancer is a manifestation of failing to detect and destroy abnormal cells. The MHC family is divided into two primary classes: MHC-I and MHC-II. Figure 1.1 shows a general overview of the

MHC family. The parts that are relevant for this thesis are shown in green.

MHC-I MHC class I molecules are present on all nucleated cells. They present peptide fragments of proteins from within the cell. If the cell is healthy, so called self-antigens are presented on the surface. This means that no foreign object is inside the cell and therefore the cell is healthy. If the cell is infected, MHC-I will present the pathogen-antigen peptides. MHC-I derives these peptides from cytosolic proteins (cytosol is the liquid found inside cells). If a Cytotoxic T-Lymphocyte or Cytotoxic T-Cell (CTL) recognizes a pathogen-antigen peptide, an immediate immune reaction is triggered. This kills the infected cell.

MHC-II MHC class II molecules can only be found on antigen presenting cells, including but not limited to T-cells and B-cells. These cells have previously *consumed*[1] a foreign particle (e.g. a bacterial cell). After having neutralized the cell, parts of its structure are presented on the surface of the cell. This means they extract the presented peptides from extracellular proteins. When a passing Helper T-cell recognizes such a peptide, it calls upon 'specialized troops' to warn for the intruder. This prepares the body for further infestation and is the foundation of vaccinations.

Scope For this thesis research is limited to the human version of MHC called Human Leukocyte Antigen (HLA), more specifically to the three main HLA class I molecules: HLA-A, HLA-B and HLA-C. The input features are limited to aminoacid sequences for both the MHC antigens and the peptide sequences. Extra data such as 3D structures are not taken into account since this would make the tool impractical for actual usage. The peptides to be considered were originally limited to sequences of length 9, since this is by far the most common sequence length of peptides found

[1] endocytosed and then digested into lysosomes

binding to MHC-I proteins. Later on also peptides of length 10 were added.

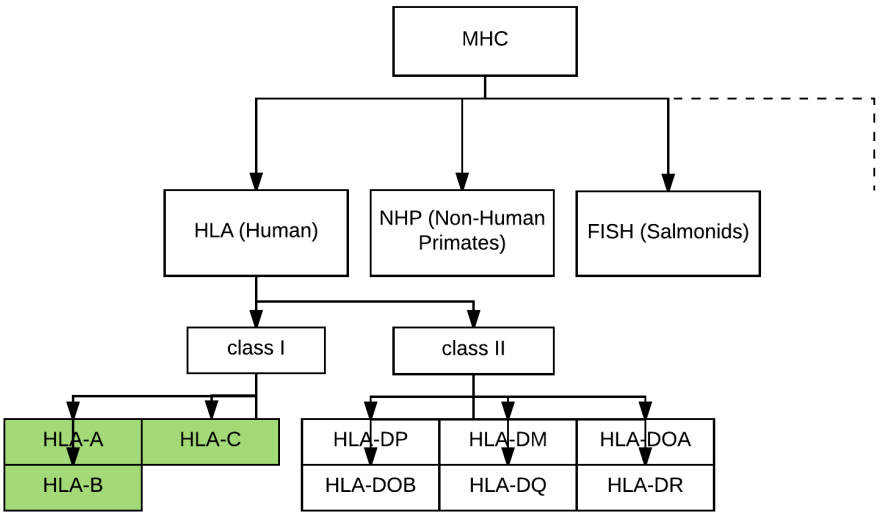


Figure 1.1: *MHC family overview*

1.3 Use case

Researchers at our university work with the hospital UZA in Wilrijk, where data is anonymously collected from patients. The blood of these patients will be used to performed. In order to achieve as much succes as possible, the researchers already use a predictive tool. Unfortunately these still suffer from inaccuracies that might be reduced further. Adding a very different second tool will inevitably yield different results. Taking the intersection of the positive results could greatly increase the chances of success for the researchers.

Chapter 2

Background and related work

In this chapter we will give some background information on immunology and neural networks. The goal of this chapter is to enable readers without knowledge of either of these subject to understand the contents of the thesis, hopefully making the research interesting for a more general audience and not just for bioinformaticians.

2.1 Immunology

Immunology is a very complex subject on which many books have been written and courses given. The goal of this section is to allow readers without any knowledge of immunology to understand how the (human) immune system works on a very general level. We believe this is needed to illustrate the importance and scale of the problem domain.

To defend itself from outside threats, the human body has several different defense systems in place:

External barrier This is the first layer of protection the human body uses to protect itself (e.g. skin, mucuous, ...). After these barriers are breached,

bacteria or virusses can find their way into the human body. At this point, the cells within the body start to work to protect the host.

Internal defense The internal defenses can be categorized into two large systems: The innate (or non-specific) defense system and the adaptive (or specific) defense system. As the name suggests, the innate defense system is a general defense system that attempts to rid the body of any intruder. Should this innate defense system fail, the body can call on the adaptive defense system for backup.

Innate defense system This general type of defense consists of many types of cells. In this paragraph we will consider the two important types of cells most relevant for this thesis: the phagocytes and the Natural Killer Cells (NK cells):

- Phagocytic cells look for intruders and consume them. After having consumed the pathogens, small parts of these pathogens (peptides) are presented on the membrane of the phagocytic cell using MHC-II.
- NK cells roam the blood and lymph looking for abnormal cells. These cells are unique in the sense that they can kill the body's own cells if these are infected with a virus or have become cancerous. When an NK cell encounters another cell, it checks the presented peptides on the cell. If the peptides appears to be a self-antigen, the NK cell will proceed. If however the peptide is a pathogen-antigen, the cell is recognized as infected and will be killed.

Adaptive defense system The adaptive defense system has to be introduced to pathogens before it can recognize and attack them. Once the adaptive defenses have been introduced to a certain pathogen, they remember it. This is the main difference between innate and adaptive systems. It is also the reason why vaccines work.
The adaptive defense system uses two complementary techniques: humoral immunity and cellular immunity.

- Humoral immunity works on a relatively large scale. B lymfocytes roam the blood and lymf and contain unique antibodies on their membrane. After finding intruders, they bind onto them, multiply and release antibodies. This creates a chainreaction that eventually leads to the death of all intruder cells.

- Cellular immunity occurs when pathogens get inside the cells. T lymfocytes come in several forms. The most important are the Helper T cells and Cytotoxic T-cells (CTLs). T-cells can only recognize pathogens when presented on MHC (both class I and II). Helper T cells bind to specific MHC-II/antigen combinations and essentially raise the alarm. CTLs are the actual killers, they check MHC-I/antigen combinations. If the binding is succesful, they kill the infested cell. Figure 2.1 shows an illustration of a T-cell binding to an antigen/MHC-I combination.

Auto-immune diseases such as MS or type-I diabetes happen when something goes wrong in the adaptive immune system. AIDS is a disease that interferes with the immune system as a consequence of the HIV virus. The HIV virus disables the Helper T-Cells.

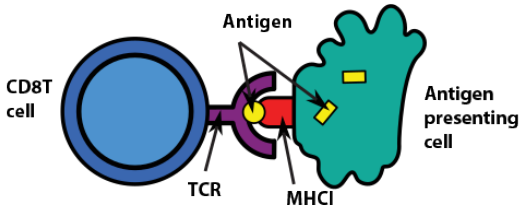


Figure 2.1: *MHC-Peptide binding*[1]

2.1.1 Proteins

The presented 'parts', pathogens and peptides are proteins. These come in the form of sequences of aminoacids. In total about 500 aminoacids are known, however only 20 aminoacids are found in the genetic code.

[1]Figure downloaded from [5]

Only these 20 are relevant for this thesis. Each of these aminoacids have a name but are mostly found in literature by their three-character or single-character representation:

alanine - ala - A	lysine - lys - K
arginine - arg - R	methionine - met - M
asparagine - asn - N	phenylalanine - phe - F
aspartic acid - asp - D	proline - pro - P
cysteine - cys - C	serine - ser - S
glutamine - gln - Q	threonine - thr - T
glutamic acid - glu - E	tryptophan - trp - W
glycine - gly - G	tyrosine - tyr - Y
histidine - his - H	valine - val - V
isoleucine - ile - I	asparagine/aspartic acid - asx - B
leucine - leu - L	glutamine/glutamic acid - glx - Z

The attentive reader may notice that the list[2] contains 22 items instead of the afore mentioned 20. In some cases it is not possible to differentiate between two aminoacids (for example between asparagine (N) and aspartic acid (D)). The uncertainty between two aminoacids is shown with a special symbol illustrative of the specific uncertainty (asx or B in this example).

2.2 Artifical Neural Networks

A popular technique in machine learning, and the technique used in this thesis is the Artificial Neural Network (ANN). In its most basic form, an ANN is composed of neurons that are connected to each other. As can be seen in Figure 2.2, each ANN has three general types of layers:

- The input layer
- The hidden layer(s)

[2]List downloaded from [6]

- The output layer

The input layer is (predictably) the layer that receives the variables or features of a given problem. These inputs are then propagated through the hidden layers to the output layer. The hidden layers perform a series of (non)linear functions that convert the input data into data that the output layer can use. This conversion is often called feature extraction since the network learns which input data is more important than others.

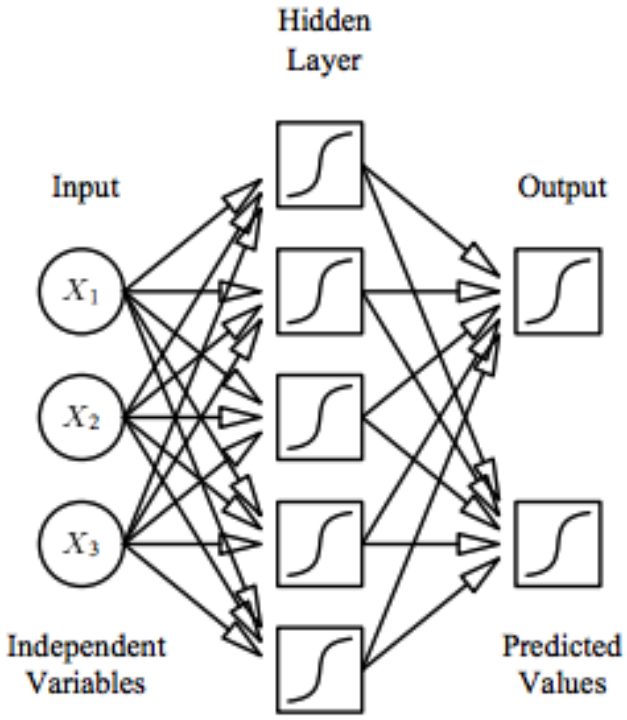


Figure 2.2: *A neural network*[3] *with a single hidden layer.*

2.2.1 Feedforward Neural Network

A feedforward neural network is the most basic neural network[8]. Much as the name suggests it is a network where the information flows in a single direction. Information always moves from one layer to the next, starting at the input nodes, through the hidden nodes and ending in the output nodes. By definition it can't contain any loops or cycles.

[3]Downloaded from [7]

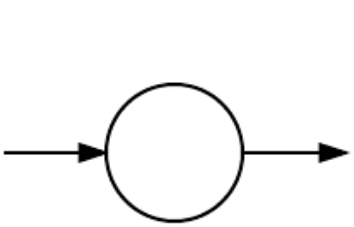
In each node, an activation function determines the output of the node based on its inputs. Neural networks using non-linear activation functions and with a depth of at least one hidden layer can even be proven to be universal function approximators[9].

2.2.2 Convolutional Neural Network

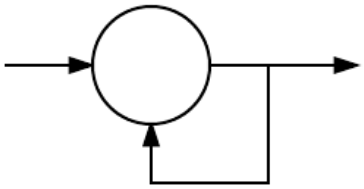
A Convolutional Neural Network (CNN) is a type of deep feedforward neural network. CNNs are special in that they use *convolutional* layers and *pooling* layers. The great strength of CNN is their ability to recognize spatial patterns. This makes them the default for analyzing images. The convolutional layer(s), when applied on a RGB image, applies a filter on a small part of the input along the x and y axis, but spans the entire depth of the input. For example: feeding an RGB JPEG image of 100×100 pixels to a CNN would need an input layer with dimension $100\times100\times3$ (100 pixels wide, 100 pixels high, 3 color channels deep). A filter of size 2×2 works on blocks of 2 pixels wide, 2 pixels high and all 3 channels deep. This filter is then slided over the entire image with a stepsize called the stride. This sliding filter yields a 2-dimensional result. Adjacent results are then combined using pooling layers. This allows a CNN to recognize patterns such as lines in an image. The deeper layers in state-of-the-art CNNs have been able to recognize structures (e.g. wrinkels) or objects (e.g. eyes, faces, noses)[10].

2.2.3 Recurrent Neural Network

A Recurrent Neural Network (RNN) is a special type of neural network because (unlike feedforward neural networks) it contains directed cycles. RNN layers use the output of the network at a certain iteration as input for a next iteration. This recursive aspect of RNNs make them extremely useful for sequential data, especially if the output of the network for a certain element depends on both that element and the output of the previous element(s). When drawn, a RNN node is represented as a cyclic graph (see Figure 2.3b). This cyclic graph can be *unfolded* into a non-cylcic graph



(a) *A basic feedforward NN node uses its input(s) and then propagates the output further through the network.*



(b) *A RNN node takes an input (often an element from a sequence) and generates output. This output is used as (extra) input for the next element in the sequence.*

Figure 2.3: *An example of a classic feedforward neural network node and a recurrent neural network node.*

where each input stands for the next element of some sequence. This unfolding is shown in Figure 2.4.
Typical usecases for RNNs are stock prediction or text analysis because in both cases the output at a certain point could be (partially) dependent on previous outputs. Depending on the problem at hand, a RNN layer can have one or more inputs and one or more outputs:

- **one-to-one** The RNN has one input and one output. This is the same as a basic NN (e.g. image classification)
- **one-to-many** A single input generates an output sequence (e.g. image captioning)
- **many-to-one** Multiple inputs yield a single output (e.g. classify text into *positive* or *negative* sentiment)
- **many-to-many** Multiple inputs yield multiple outputs (e.g. translation of a sentence between two languages)

2.2.4 Training neural networks

The nodes between layers are connected with weighted edges. When initializing a neural network, random values are assigned to these various

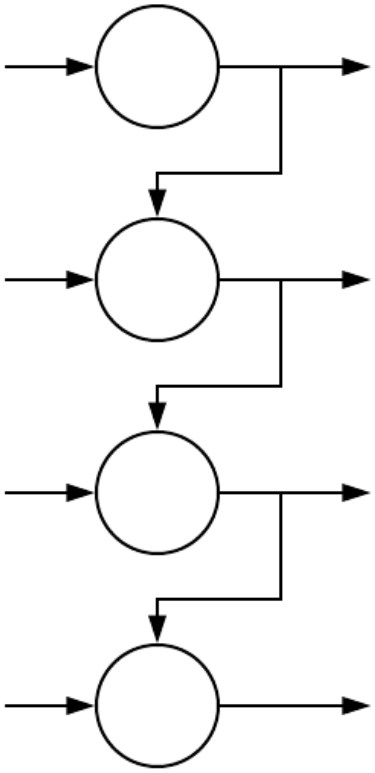


Figure 2.4: *A RNN can be unfolded by writing it out for multiple elements in the sequence. A RNN operating on a sequence of 4 elements, could be unfolded as a 4-layer NN. In this example, the RNN also has 4 outputs.*

weights throughout the network. The process of finding the (locally) optimal weights is called *training the network*. This is achieved by minimizing a cost function. A popular cost function is the sum of squared errors:

$$J = \sum (y_i - \hat{y}_i)^2$$

where y_i is the actual label of element i and $\hat{y}_i$ is the predicted label of element i. Neural networks are often trained using a technique called *gradient descent* with *backpropagation* which will be explained below.

Gradient descent As stated above, the goal of training a neural network is to find the optimal weights, i.e. weights that yield the lowest cost. Due to the *curse of dimensionality*[11] brute force algorithms are often not an option. A very efficient heuristic is gradient descent. Gradient descent is an optimization algorithm that uses the derivative of the cost function to determine the direction of its minimum. If the derivative of the cost function in a certain point is negative, the cost function is going downhill and the

algorithm will maintain its current direction. Vice versa if the derivative is positive, the cost function is going uphill and the minimum is in the other direction. This allows the algorithm to *descent* the cost function very quickly, with the downside of not being globally optimal (the algorithm might converge to a local minimum).
If we write the neural network as a function, the output of the network is given by $\hat{y} = f(W, X)$. Where W are the weights of the network and X is the vector of input features.
We define the cost function J as:

$$\begin{aligned} J &= \sum \frac{1}{2}(y - \hat{y})^2 \\ &= \sum \frac{1}{2}(y - f(W, X))^2 \end{aligned}$$

Gradient descent will converge when the derivative of the cost function

$$\begin{aligned} \frac{dJ}{dW} &= \sum f'(W, X) \\ &= 0 \end{aligned}$$

It reaches this state by iteratively adjusting the weights by subtracting a stepsize multiplied with the derivative of the costfunction w.r.t. the weight matrix. For the network seen in Figure 2.5, this is illustrated in Equations 2.1 and 2.2.

$$W^1 = W^1 - stepsize \cdot \frac{\delta J}{\delta W^1} \tag{2.1}$$

$$W^2 = W^2 - stepsize \cdot \frac{\delta J}{\delta W^2} \tag{2.2}$$

Gradient descent comes in three versions:

1. **Batch Gradient Descent** uses the entire dataset available to compute the gradient of the cost function per update. This makes batch gradient descent very slow and memory intensive.

2. **Stochastic Gradient Descent** performs an update per data sample. This makes it a very fast algorithm, but unfortunately it also makes the algorithm *overshoot* the optimum. This is often mitigated by slowly reducing the stepsize.

3. **Mini-batch Gradient Descent** receives batches of a limited number of samples. This takes the best of both worlds since it limits the variance of the error (because it uses multiple samples), but without recalculating the gradient for all samples (many of which are quite similar to eachother).

Backpropagation Backpropagation is a technique that calculates the derivative of the cost function w.r.t. the weight matrices. It is conceptually very simple since it simply applies the chainrule over and over.
Notice that the derivative of the cost function w.r.t. the last weight matrix (W^2 in Figure 2.5) depends only on the node from which it originates and the output of the network. The derivative w.r.t. the second to last weight matrix (W^1 in Figure 2.5) depends on the originating node, the output and the last weight matrix (W^2). Basically, backpropagation computes the derivative of the 'last' weight matrix first, then uses this while backtracking through the entire network. This also explains the name of the algorithm.

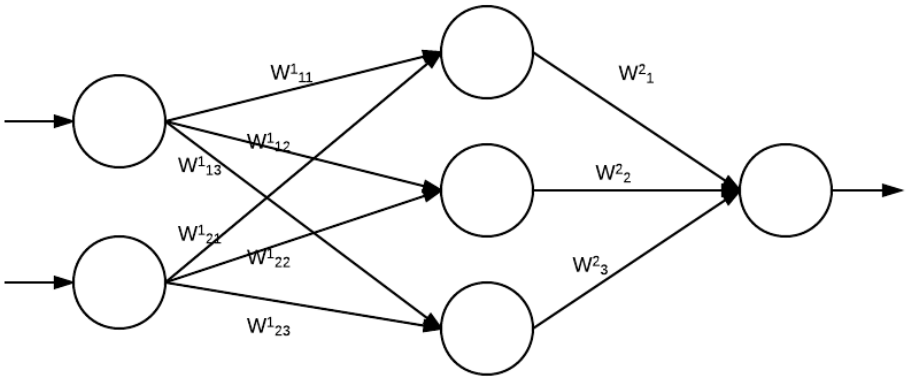


Figure 2.5: *A neural network with two weight matrices*

Chapter 3

Methodology

3.1 Technologies and Tools

This section gives an overview of the tools and libraries used throughout the various parts of this thesis. All code was written in Python (version 3.6.2).

Keras (2.0.6) is an open-source, high-level neural networks API written in Python. It can be configured to use Tensorflow, CNTK or Theano as backend. We used the default backend which is Tensorflow. Keras greatly simplifies the code needed to design, train and evaluate Neural Networks, while retaining the performance of more complicated libraries. This makes it a great tool for research.

Tensorflow (1.2.1) was used as the underlying library for keras. It is an open-source library for numerical computations such as neural networks. It was originally developed by the Google Brain Team to conduct machine learning and deep neural network research.

BioPython (1.70) is a set of tools for biological computations, written in Python. In this thesis we used the *SeqUtils* and the *SubsMat* mod-

ules. The *SeqUtils* module was used to access data from aminoacids. The *SubsMat* module contains the substitution matrix used for encoding the aminoacids.

NumPy (1.13.1) is a package for scientific computing in Python. It is part of the **SciPy (0.19.1)** ecosystem. Its N-dimensional array structure is used as the main data container for the various mathematical libraries used throughout the thesis.

PyDot (1.2.3) with **graphviz (0.8)** python interface is a useful library that can generate an architectural image of a Neural Network in Keras. It is not only useful for documentation, but also a very handy debugging tool.

GraphViz (2.40.1) is an open-source graph visualization software. It is used by the PyDot library to generate images of a Keras model.

Plotly (2.2.1) is a Canadian based online data and visualization tool. It comes with a very easy-to-use Python wrapper. Its ease of use is the main reason why it was chosen for this specific project.

Clustal Omega is a tool found on the EBI website[1]. Clustal Omega performs *multiple sequence alignment* on three sequences or more and is limited to a maximum of 4000 sequences or a filesize of 4 MB.

All reading, parsing and validating of files was done manually (i.e. without help of external libraries). We don't recommend this as it is a very time consuming process. For interested readers we propose the **Pandas** library[2] instead.

[1] https://www.ebi.ac.uk/Tools/msa/clustalo/
[2] https://pandas.pydata.org/

3.2 Data sets

The data used is the result of a merge from two different datasets. From EBI we collected the MHC details. From IEDB we collected experimental binding data. These two sets were then combined to create a single set containing only relevant data (see Subsection 3.3.3 on details as to how they were combined).

3.2.1 EBI

EMBL-EBI is part of the European Molecular Biology Laboratory (EMBL). EMBL was the first to create a nucleotide sequence database in 1980 in Germany. They extracted information from scientific literature and have now evolved into a hub with a huge amount of contributions from researchers worldwide. In 1992 EMBL expanded into EMBL-EBI (EMBL-European Bioinformatics Institute) with two databases:

- European Nucleotide Archive, which contains the original Nucleotide Sequence Database.
- Uniprot, originally known as Swiss-Prot–TrEMBL

Currently EMBL-EBI still provides a wide range of tools and databases, but also performs its own research and even offers training programs.
For this thesis, the PD-IMGT/HLA Database was used to collect details from the HLA molecules. At the time of writing this thesis, the database contains 17,509 allele sequences with detailed information on the material from which the sequences were collected and the data on the validation of the sequences. It is mainly maintained by people from Anthony Nolan research center[3], with help from people from EBI and Stanford University Medical School[4].

[3]https://www.anthonynolan.org/
[4]https://web.stanford.edu/group/parhamlab/

3.2.2 IEDB

The Immune Epitope Database and Analysis Resource (IEDB) offers an extensive amount of experimental data. IEDB offers users easy acces to this data using extensive search tools and offers users acces to various other tools. IEDB is funded by the National Institute of Allergy and Infectuous Diseases (NIAID).
After searching for the specific data we needed, we unfortunately were unable to download it using the interface provided. Thankfully IEDB also provides its users with the raw datafiles[5]. These had to be manually cleaned and pruned, details on how this was done are written in Subsection 3.3.2.

3.3 Data preprocessing

3.3.1 MHC

We downloaded the MHC details from EBI [6]. The relevant data was found in *A_prot.fasta*, *B_prot.fasta* and *C_prot.fasta*. These contained the proteins name, type and length as well as the full aminoacid sequence. A first step was to remove any entry with invalid data (such as sequences shorter than 365 or *non-coding* sequences[7]) or sequences with uncertainty in the aminoacids (As explained in Subsection 2.1.1 symbols exist to express uncertainty of which aminoacid is present). Sequences like this were rare enough that we chose to remove them from the dataset.
Even after removing many of these entries, MHC sequences still come in a wide variety of lengths. The next step in preprocessing was to fix this discrepancy. Initially we padded the sequences with a special symbol. The model should then be able to learn that these symbols are meaningless.

[5]These can be found at `http://www.iedb.org/database_export_v3.php`

[6]Downloaded on July 10th, 2017 from `ftp.ebi.ac.uk/pub/databases/ipd/imgt/hla/fasta`

[7]'non-coding' means that the MHC protein is invalid, this is labeled in the database using the letter N after the MHC name (for example: HLA-A*01:123N)

Later on, we isolated the relevant subsequence within each MHC protein. Since peptides only bind to specific parts of the MHC protein we aligned several sequences and isolated a semi-constant pre- and postfix. To perform this alignment we randomly shuffled all the MHC proteins. Since the multiple sequence alignment tool allows up to 4000 sequences, the set was then sliced in 4 parts of roughly equal size. The first part was used to perform the alignment. The shuffling assures that sequences from many different proteins occur in this set of 4000 sequences, eliminating the danger of having an alignment biased towards a proteins that occur more in the first parts of the database. Figures 3.1a and 3.1b show what (a part of) the alignment looked like and which pre- and postfix was chosen[8]. These subsequences would serve as delimiters for the relevant part of each MHC protein. This had two benefits: The dimensionality of the problem was greatly reduced (MHC sequences of 365+ aminoacids were reduced to sequences of 116 aminoacids) and the sequences that were originally of various lengths, now have a fixed length (which is a requirement for NN training) without the need of padding.

3.3.2 Peptides

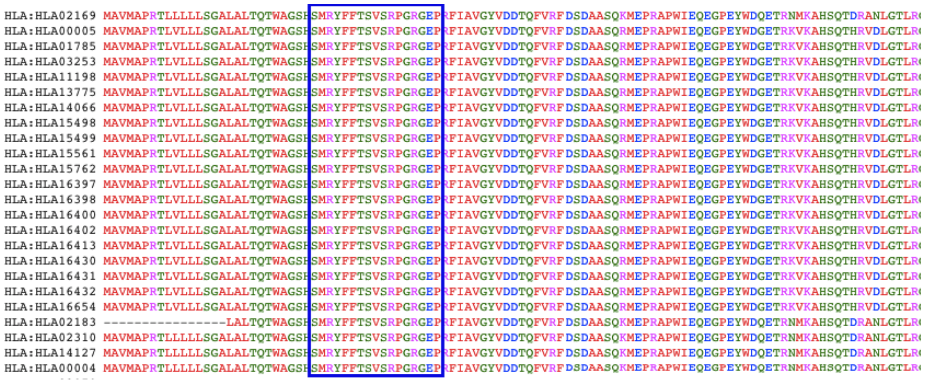
We downloaded the mhc_ligand_full.csv file from IEDB [9]. This dataset contains a lot of information that was irrelevant for our research. When parsing the file, we kept only 4 fields (out of 98):

1. **Allele description** This field contains the peptide aminoacid sequence. It will be used as one of the two inputs in the neural network.

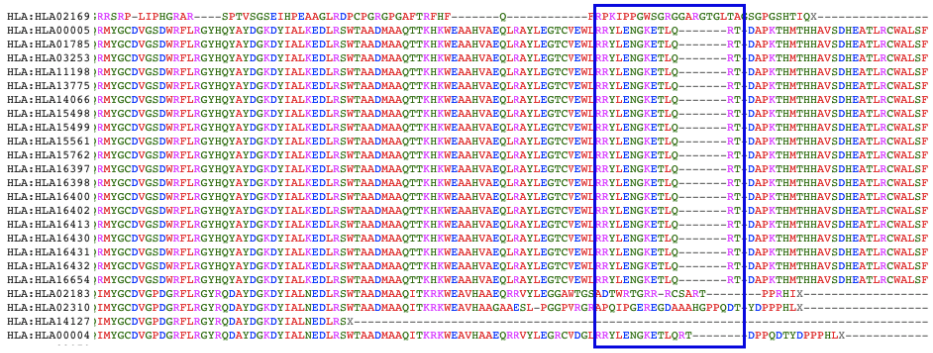
2. **Qualitative measure** The qualitative measure classifies the result of binding experiment and will be used to label the experiments. Possible values are *negative*, *positive-low*, *positive-intermediate*, *positive-high* or *positive*.

[8]The entire alignment of the sequences used can be requested by mail from bruno.dedeken@student.uantwerpen.be

[9]downloaded on July 10th, 2017 from `http://www.iedb.org/database_export_v3.php`



(a) *prefix*



(b) *postfix*

Figure 3.1: *Constant pre- and postfix (**(a)** and **(b)** respectively) delimiting the relevant subsequence for MHC-peptide binding. This was determined by manually finding (relatively) constant parts after performing multiple sequence alignment on approx. 4000 randomly selected sequences.*

3. **Allele names** These contain the names of the different MHC proteins. This will be used as key to combine the data with the MHC details from the EBI dataset.

4. **MHC allele class** As explained in Section 1.2, we will only work on MHC class I. The MHC allele class is used to filter all elements of this class.

3.3.3 Combination

After having parsed and cleaned the data, both sets were joined using the MHC names. The resulting fields are:

1. The MHCs name: The name of the MHC molecule (e.g. HLA-A*01:01) is kept so we retain the possibility to filter the different types of HLA molecules (e.g. filter all HLA-A sequences or only the HLA-A*01 family).

2. The peptide sequence: A short sequence of 9 or 10 aminoacids that will be used as one input in the neural network.

3. The MHC sequence: A sequence of 116 aminoacids. Originally the available sequences were of various different lengths but because of the isolation of the relevant part of the MHC protein, all sequences have a fixed length. This sequence will be the second input in the neural network.

4. The resulting binding preference: The resulting binding preferences *Negative* or *Positive* are encoded as *0* or *1* resp., this way we've already prepared the labels for binary classification training.

First a selection had to be made of what was considered *positive* and what was considered *negative*. Not all submissions of the experiments have adhered to the same naming convention w.r.t. the binding result. This resulted in two types classification: Some researchers had classified

their results using the more detailed labels *negative*, *positive-low*, *positive-intermediate*, *positive-high* others merely used the labels *negative*, *positive*. To generalize, we converted all labels containing 'positive' into *positive* labels. This meant the loss of some information, but was deemed a necessary loss in order to maintain as much uniform data as possible. A histogram of the categories pre-generalization can be seen in Figure 3.2.

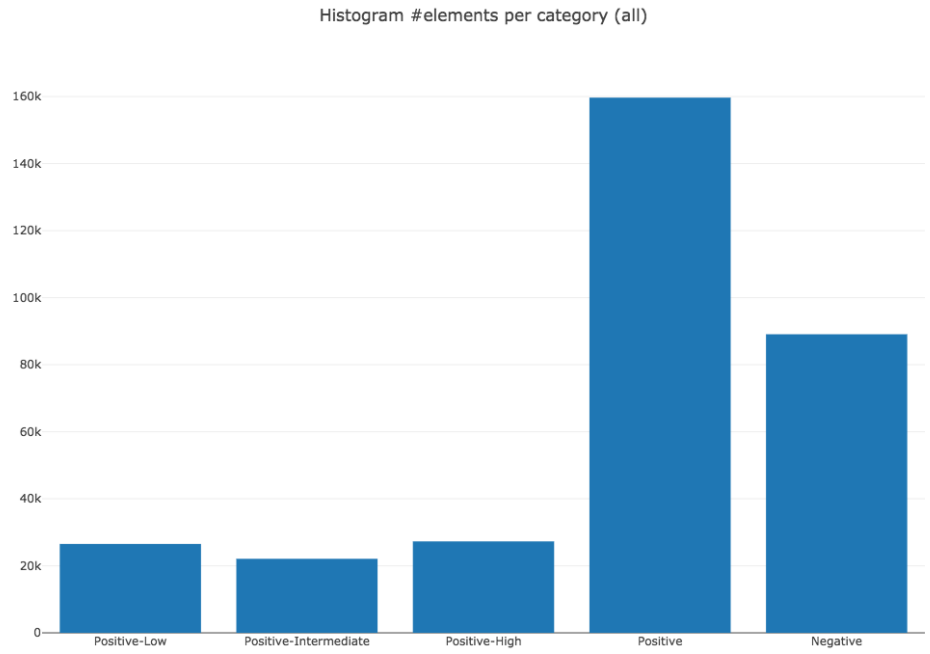


Figure 3.2: *Histogram per classification category before generalization.*

3.4 Data representation

The data used by the neural network consists of two sequences of aminoacids. Each aminoacid is symbolized by one of 20 characters. The encoding of these aminoacids can be done in any of several ways. An obvious first choice is to create one-hot vectors of size 20, where each aminoacid would have value 1 on the index corresponding to its own index (e.g. the first aminoacid would be encoded as [1 0 0 0 ...], the second aminoacid would be [0 1 0 0 0 ...] and so forth). However, we chose to encode each aminoacid in a feature vector of size 20 using a BLOSUM matrix.

This is a substitution matrix where the value M_{ij} is related to the observed number of evoluationary changes of aminoacid i into aminoacid j. In this technique we first replaced each aminoacid by a vector of size 20 corresponding to the matching column from the BLOSUM90[10] matrix, then we normalized all values in the matrix. A (non-normalized) example can be seen in Figure 3.3

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z
A	5	-2	-2	-3	-1	-1	-1	0	-2	-2	-2	-1	-2	-3	-1	1	0	-4	-3	-1	-2	-1
R	-2	6	-1	-3	-5	1	-1	-3	0	-4	-3	2	-2	-4	-3	-1	-2	-4	-3	-3	-2	0
N	-2	-1	7	1	-4	0	-1	-1	0	-4	-4	0	-3	-4	-3	0	0	-5	-3	-4	4	-1
D	-3	-3	1	7	-5	-1	1	-2	-2	-5	-5	-1	-4	-5	-3	-1	-2	-6	-4	-5	4	0
C	-1	-5	-4	-5	9	-4	-6	-4	-5	-2	-2	-4	-2	-3	-4	-2	-2	-4	-4	-2	-4	-5
Q	-1	1	0	-1	-4	7	2	-3	1	-4	-3	1	0	-4	-2	-1	-1	-3	-3	-3	-1	4
E	-1	-1	-1	1	-6	2	6	-3	-1	-4	-4	0	-3	-5	-2	-1	-1	-5	-4	-3	0	4
G	0	-3	-1	-2	-4	-3	-3	6	-3	-5	-5	-2	-4	-5	-3	-1	-3	-4	-5	-5	-2	-3
H	-2	0	0	-2	-5	1	-1	-3	8	-4	-4	-1	-3	-2	-3	-2	-2	-3	1	-4	-1	0
I	-2	-4	-4	-5	-2	-4	-4	-5	-4	5	1	-4	1	-1	-4	-3	-1	-4	-2	3	-5	-4
L	-2	-3	-4	-5	-2	-3	-4	-5	-4	1	5	-3	2	0	-4	-3	-2	-3	-2	0	-5	-4
K	-1	2	0	-1	-4	1	0	-2	-1	-4	-3	6	-2	-4	-2	-1	-1	-5	-3	-3	-1	1
M	-2	-2	-3	-4	-2	0	-3	-4	-3	1	2	-2	7	-1	-3	-2	-1	-2	-2	0	-4	-2
F	-3	-4	-4	-5	-3	-4	-5	-5	-2	-1	0	-4	-1	7	-4	-3	-3	0	3	-2	-4	-4
P	-1	-3	-3	-3	-4	-2	-2	-3	-3	-4	-4	-2	-3	-4	8	-2	-2	-5	-4	-3	-3	-2
S	1	-1	0	-1	-2	-1	-1	-1	-2	-3	-3	-1	-2	-3	-2	5	1	-4	-3	-2	0	-1
T	0	-2	0	-2	-2	-1	-1	-3	-2	-1	-2	-1	-1	-3	-2	1	6	-4	-2	-1	-1	-1
W	-4	-4	-5	-6	-4	-3	-5	-4	-3	-4	-3	-5	-2	0	-5	-4	-4	11	2	-3	-6	-4
Y	-3	-3	-3	-4	-4	-3	-4	-5	1	-2	-2	-3	-2	3	-4	-3	-2	2	8	-3	-4	-3
V	-1	-3	-4	-5	-2	-3	-3	-5	-4	3	0	-3	0	-2	-3	-2	-1	-3	-3	5	-4	-3
B	-2	-2	4	4	-4	-1	0	-2	-1	-5	-5	-1	-4	-4	-3	0	-1	-6	-4	-4	4	0
Z	-1	0	-1	0	-5	4	4	-3	0	-4	-4	1	-2	-4	-2	-1	-1	-4	-3	-3	0	4

Figure 3.3: ***BLOSUM90 matrix****Using the BLOSUM90 matrix, the feature vector for aminoacid A would be:* $[5\ -2\ -2\ -3\ -1\ -1\ -1\ 0\ -2\ -2\ -2\ -1\ -2\ -3\ -1\ 1\ 0\ -4\ -3\ -1\ -2\ -1]^T$. *Aminoacid H would have feature vector* $[-2\ 0\ 0\ -2\ -5\ 1\ -1\ -3\ 8\ -4\ -4\ -1\ -3\ -2\ -3\ -2\ -2\ -3\ 1\ -4\ -1\ 0]^T$

[10]BLOSUM90 is generally used to compare aminoacids of closely related species, whereas BLOSUM50 would be used to compare sequences of more distantly related species.

We chose to use BLOSUM90 since it is designed to work with closely related species. Since we limit ourselves to HLA molecules (as described in Section 1.2), this seemed an appropriate choice. The downside to using a substitution matrix is the large increase in complexity. The big advantage however is that similar aminoacids (from an evolutionary perspective) have a smaller distance between them than non-similar aminoacids in this high dimensional space.

A last encoding option was the use of the aminoacids physicochemical properties. The properties that we can use are the aminoacids *basicity*, *hydrophobicity*, *helicity*, *mutation stability*, *mass* and *electorchemical charge*.

3.5 Model

The neural network we wanted to design needed two inputs. The first input will consist of a peptide sequence, the second input will consist of an MHC sequence. The model has to learn the relevant features per input sequence. After having found these feature vectors, they are concatenated to create one large feature vector. This feature vector is then used to train the model for binary classification. Conceptually the model has to look like Figure 3.4. To fully create an ANN some important design choices

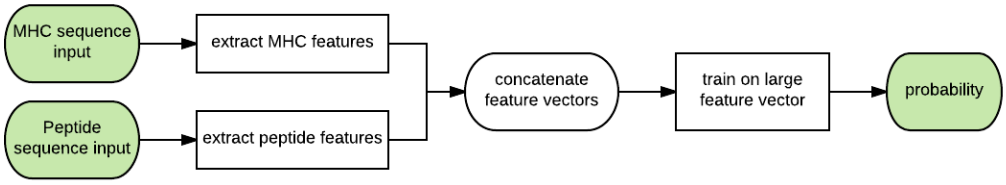


Figure 3.4: *Conceptual design of ANN*

needed to be made. These choices can be summarized as choosing the

1. types of layers
2. number of nodes per layer (width of network)
3. number of layers (depth of network)

4. activation functions

The rest of this section will elaborate on how these choices were made.

3.5.1 Design

Types of layers

As was explained in Section 2.2 we have the choice of three types of neural networks: The (classic) feedforward neural network, the recurrent neural network and the convolutional neural network. When we consider the task at hand: analysing two sequences of aminoacids in order to predict a possible reaction between them, it is extremely likely that dependencies between the different aminoacids exist.
Feedforward neural networks can't handle sequences of elements. If we wanted to proceed with this type of network, we would have to encode the entire sequences as a single (atomic) input. This is defnitely a possibility, but may not be the most ideal solution.
Convolutional neural networks on the other hand excel in finding structural information. Since our data is essentially a sequence and not a matrix, little relevant structural information is available. Of course we can convert these sequences into matrices (we did just that when encoding the aminoacids using the BLOSUM90 substitution matrix), this does not guarantee any dependencies between adjacent elements within the matrix.
Finally, recurrent neural networks use the output of a certain element in the sequence as an extra input when handling the next element of that sequence. This allows the network to learn dependencies between elements of the same sequence. This is exactly the type of situation we are in. We presume that sequential features, such as the order or (relative) position of certain aminoacids in the proteins sequence will influence the reaction between these proteins. This is the type of dependencies we want to discover. For this reason choosing a recurrent neural network seems to be the more promising strategy for our problem.

The keras library offers three types of recurrent neural network layers: SimpleRNN, LSTM and GRU.

- **SimpleRNN** is a basic, fully connected RNN. The output of the node(s) are fed back to the input, internally the nodes have a single NN with a simple activation function (i.e. tanh). The problem that arises with these kinds of RNN is that the gradient of the loss function approaches zero over time[11]. This mainly poses an issue when learning dependencies between elements over long distances in large sequences.

- **LSTM** (Long Short Term Memory) is a more sophisticated type of RNN. A LSTM node is in itself an ANN. A LSTM node consists of a memory cell and three activation functions. The three activation functions work as gates on the input, output and memory. They are called the *input gate*, *output gate* and *forget gate* respectively. These gates have a sigmoid or tanh activation function (yielding a value between 0 and 1), this way they control the contribution of the respective input, calculated output and stored memory to the output of the node[12].

- **GRU** (Gated Recurrent Unit) is a modified version of LSTM with two gates (called the *update gate* and *reset gate*). It also doesn't store a memory cell. The output of the node is an interpolation between the previous activation and the candidate activation. The update gate determines by how much the unit is updated compared to the previous activation. The reset gate is used to calculate the candidate activation.[13]

Since the MHC sequences are quite large, being able to discover relations between far away elements in the sequence is crucial. For this reason a SimpleRNN will probably not be the best choice. The better alternative will be either LSTM or GRU. We experimented with all three types of RNN

[11] This is called the Vanishing Gradient problem

[12] A clear and detailed explanation can be found at [12]

to see which performed best. We are mostly interested in the results between LSTM and GRU, using SimpleRNN in the experiment is interesting as a comparative result against the two others. A large difference in performance between the SimpleRNN vs the two others will strengthen our assumption that long distance dependencies are in fact important. After trying these types of recurrent layers we had the best results with LSTM layers (see Table 4.3) so this is what we used.
When adding extra recurrent layers, we did not see better results, for this reason we limited the amount of recurrent layers to one per input, the following layers are a combination of Dense layers and Dropout layers.

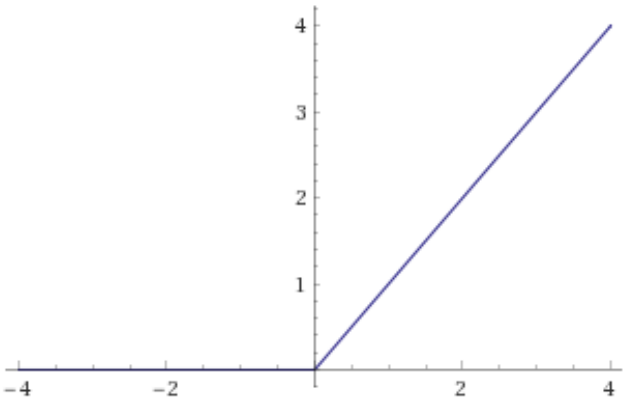
Activation functions

The first choice of activation function is which function to use in the final classifying node. For binary classification the default is the sigmoid function. The sigmoid function had our preference, but for completeness we tested the same architecture with the same data and different activiation functions on the output layer. The results of this experiment can be found in Chapter 4, Table 4.1.

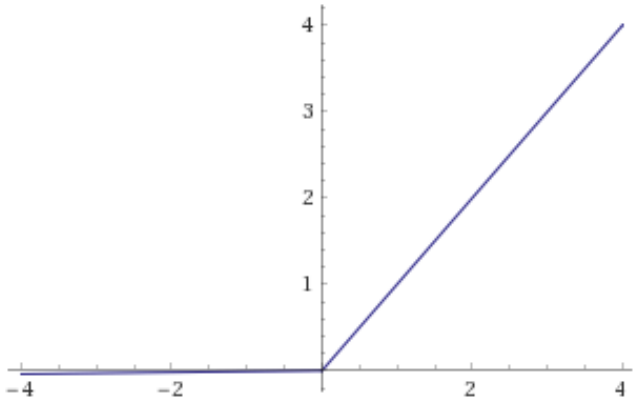
The next choice to be made was the activation function used in the hidden nodes. Similar to the methodology explained above we tested several activation functions, the results of this experiment are listed in Table 4.2 in Chapter 4.

Our predictive model will be used in medical research where it will be used to determine which potential reactions will lead to the most interesting experimental results. Due to the cost involved in performing these experiments, precision is more important than recall. We proceed with a leaky Relu function[13] as activation function in the hidden layers and a sigmoid[14] as binary classification function on the output node.

[13]The Leaky Relu function is explained in Figure 3.5.
[14]Figure 3.6 explains the sigmoid function.



(a) *The ReLu activation function (or Rectifier Linear Unit) is a rather simple function:* $\mathbf{f(x) = max(0, x)}$. *This function is indifferentiable in the point (0,0) which, theoretically, is a problem during trainig (see Subsection 2.2.4). In practice however this doesn't lead to any issues. Since the function is differentiable in any point NOT equal to (0,0), the problem only becomes apparent if the derivative in that exact point has to be calculated. Chances of this happening in real-life training are small enough not to be taken into account.*



(b) *Leaky Relu is a variant of the Relu activation function. In leaky Relu, the graph goes below 0 for values of* $x < 0$. *The reason for this adjustment is that in classic Relu the gradient for* $x < 0$ *is always equal to zero. This means that the neurons in this situation stop responding to any change in error. To solve this problem, the derivative of all points in this range have to be non-zero. Leaky Relu solves this problem by slightly adjusting the Relu definition:* $\mathbf{f(x) = max(0.01x, x)}$

Figure 3.5: *Relu and LeakyRelu compared*

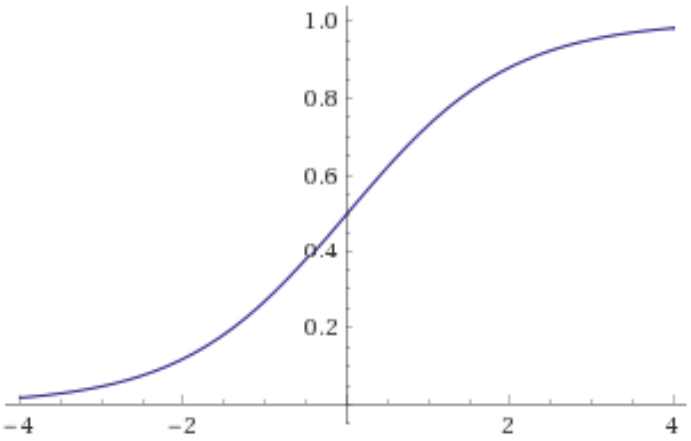


Figure 3.6: *The sigmoid activation function ($\mathbf{f(x)} = \frac{\mathbf{1}}{\mathbf{1+e^{-x}}}$) is widely used for (binary) classification. This is mainly due to two reasons:*
1) *The output of the activation function is bounded to the range [0,1]. This means that the output is always a probability of a datapoint belonging to class A ($P(A) = \sigma(x)$) or class B ($P(B) = 1 - \sigma(x)$).*
2) *The curve around $x = 0$ is quite steep. This means that the sigmoid function reacts strongly to changes in the inner region. This makes predictions for one or the other class stronger.*

3.5.2 Architecture

We started the design of the neural network with one LSTM and one Dense layer per input *subnet*. After concatenating these, one extra Dense layer was added before the final classification layer. This was the foundation on which we would have to build. This foundation quickly led to what we call a Wide and Shallow (WS) version of the network. As an extra, we also created a Deep and Narrow (DN) version. The idea behind this was that the increased depth of the network had to enable it to learn the most important features on a higher level, while the narrower design would reduce overfitting.
Overfitting has proven to be quite a difficult challenge we had to deal with throughout this project. In an attempt to decrease overfitting we experimented with Dropout layers [14], which proved to be very effective.

In a second phase we expanded both the WS and DN network architectures. The main changes done in this second version was the increased width of the layers. We did this for both versions and found that an increased width yields good results in the shallow version, but not so for the deep version. Results are shown in Chapter 4.

In a final design phase we updated both the WS and DN architectures again. We can summarize these changes as a middle road between the previous two versions. We mainly re-narrowed the layers and fine-tuned the dropout rates on the various layers. These final versions yielded the best results so far and performed to our satisfaction.
After noticing that both these architectures were quite complex, out of interest we went back to the first (basic) version and decided to improve this version while keeping it as simple as possible. This design will be referred to as the Simple Model (SM) and was created with the idea of using as little layers as possible.

3.5.3 Training

In the previous subsection, we have explained the type and structure of the data, the way it has been preprocessed and how the neural network is designed. This subsection will explain the process that was used to train the neural networks.

Data sets

The first part before training a neural network (or any other type of model) is to divide the data in a training set and a test set. Initially the data was divided randomly in a 90% training set and a 10% test set. This gave impressively good results. After some time however we learned that the predictive capacity of the model for a protein was highly dependent on how well it had already learned the features of that specific protein. If for example the network had seen data using the protein *HLA-B*01:02*, it would be able to predict other queries with *HLA-B*01:02* much more accurately,

even for unseen peptides. Since the model has to work generically, this effectively rendered the training-testing data division useless. As far as the MHC protein sequences was concerned, the network was just testing on prelearned data.
To solve this an extra module was added that isolated all entries of specific proteins before dividing the data into training and test data. This meant that the network would be tested on proteins that were unseen during training which immediatly resulted in worse accuracy, but more useful models. In a later stage, this module was expanded with the possibility to isolate entire families or types of MHC proteins (e.g. isolate all HLA-A*01 proteins or all HLA-C proteins). This would allow us to test the generic qualities of the model on a very granular level (how good is it on unseen proteins, unseen families or unseen classes of proteins?). As an added bonus, we could also test the possibilities and effects of pretraining the model on a well known class (e.g. HLA-A) and then test the predictive qualities after retraining on a much less known class (e.g. HLA-C).

The number of epochs that was used for training was set to 30. To avoid running more epochs after convergence, a feature was activated that stopped the training if no improvement was made during 5 epochs. Unfortunately it became apparent that the model would sometimes perform worse after more epochs than before (e.g. the model performed worse after 30 epochs than after 27 epochs).
To fix this problem, models performing worse after an epoch were discarded. Only when a models performance had improved was it store and loaded at the end of the training. In order to avoid overfitting on both the training data and the test data, we divided the data in one extra subset: the validation set.
The division of training data - validation data - test data was set to 70% - 20% - 10% respectively. The training data is used solely as input for the model during the training. After each epoch, the performance of the model is measured using the validation data. Finally, the best version of the model is loaded to review its final performance on the test data. To

further improve the reliability of the performance observed, each training session is reexecuted 5 times after which the average of the various metrics is taken as the final qualitative measure.

Metrics

In the initial phase, we naively used the accuracy on the training data as the sole performance metric. Halfway the development of this project, it was obvious that a more detailed view of the performance of the model was required. To achieve this more detailed view, the precision and recall were added to the list of performance metrics. The confusion matrix was also generated to give an added insight.
Knowing these metrics enabled us to better compare the different models. Since precision was deemed slightly more important than recall, we were able to further customize the model to fulfill this request.

Imbalanced data

The data used for training/validation/testing is biased towards positive results when compared to real-world data. In the dataset that was downloaded, we had approx. 68% positive labels and 32% negative labels. In real-world data (that was given when the research period came to an end) the ratio of positive vs negative was in the range of 1.3% vs 98.7%. This difference can be explained by the selection process used to select candidates on which the experiments are performed that contribute to the online datasets. It is understandable that these experiments are done with candidates that are more likely to yield a positive result.
We tried three techniques to solve this difference:

1. **artificial data augmentation** We artificially augmented the data with negative entries to convert the ratio to roughly 25% positive vs 75% negative instances. This augmentation is done by generating random sequences of aminoacids for both inputs. Since the chances of having a positive binding result between two random sequences are

close to zero, we can label these as negative. This however didn't show any meaningful difference in performance (Table 4.8).

2. **weighted cost function** Another technique that was employed is the definition of a custom objective function. This assigns a higher weight to an underrepresented class. Unfortunately this would create an imbalance when considering the training data and test data because the imbalance is limited to the training data and the real-life data. Since the real-life data we possess are labeled using another predictive tool, we don't have any guarantees as to which labels are correct and which aren't. This means we can't use it for testing. A weighted cost function would yield invalid results on our validation and testing data sets.

3. **conversion function** The last technique that was tried (and actually used) was to create a conversion function that changes the predicted values, created on the training data set distribution, to probabilities matching the real-life distribution.

Weighted cost function

As cost function the log loss function was used. In the general case it is defined as

$$f(x) = \frac{-1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} y_{ij} \cdot log(p_{ij}),$$

with N the number of instances, M the number of classes, $y_{ij} = 1$ if instance i belongs to class j and $y_{ij} = 0$ if instance i does not belong to class j. p_{ij} is the probability generated by the model of instance i belonging to class j.
In the case of binary log loss ($M = 2$), this is simplified to

$$f(x) = \frac{-1}{N} \sum_{i=1}^{N} (y_i \cdot log(p_i) + (1 - y_i) \cdot log(1 - p_i))$$

where y_i is the actual label of instance i and p_i is the generated probability of instance i being positive (i.e. belonging to the class corresponding to $y = 1$). Since we use the sigmoid activation function on the output layer,

the output of the model is already a probability.

We couldn't use a weighted cost function to solve the problem of imbalance between the downloaded data and the real-life data. But since the model will be used for actual experiments, it was deemed important to adjust the cost function to more closely match the priorities of a research team: A negative result translates to a higher cost than a positive result. To incorporate this in the cost function, we used a weighted log loss function: The updated function is defined as

$$f(x) = \frac{-1}{N}\sum_{i=1}^{N}(w \cdot y_i \cdot log(p_i) + (1 - y_i) \cdot log(1 - p_i))$$

where w is the weighted penalty for positive results. This means that the cost of false negatives is increased by a factor w. We set w to a value between 0.0 and 1.0, this has the effect that false negatives become less costly, which in turn means that false positives become relatively more expensive. It is reasonable to assume that a negative result leads to a higher cost for researchers than a positive result since a negative result signifies a pure loss of time and financial resources. A positive result might yield some profit for the researchers in either funds or knowledge. We chose to make a negative result twice as costly as a positive result (making $w = \frac{1}{2}$).

Conversion function

Before we can define a conversion function, we determine the relationship between the distribution of the training data (X, y) and the distribution of the real data $(\hat{X}, \hat{y})$. To do this, we rewrite the chances of our model predicting $y = 1$ for a given x. In the equation below, we use the training dataset, meaning we sample some x from X (we will use $P(y)$ to denote the more correct notation $P(y = 1)$ and $P(\neg y)$ to denote $P(y = 0)$).:

$$\begin{aligned} P(y|x) &= \frac{P(x|y)P(y)}{P(x)} \\ &= \frac{P(x|y)P(y)}{P(x|y)P(y) + P(x|\neg y)P(\neg y)} \\ &= \frac{a}{a+b} \end{aligned}$$

Now we have the case where we take this same sample x from the real-life set $\hat{X}$:

$$\begin{aligned} P(\hat{y}|x) &= \frac{P(x|\hat{y})P(\hat{y})}{P(x)} \\ &= \frac{P(x|\hat{y})P(\hat{y})}{P(x|\hat{y})P(\hat{y}) + P(x|\neg\hat{y})P(\neg\hat{y})} \end{aligned}$$

The only difference between (X, y) and $(\hat{X}, \hat{y})$ is the distribution of the data, all the rest stays the same. We introduce α and β as the ratios with which the positive and negative cases respectively have been oversampled. This means we have the following relations:

$$\begin{aligned} P(x|\hat{y}) &= P(x|y) \\ P(\hat{y}) &= \alpha P(y) \\ P(\hat{\neg y}) &= \beta P(\neg y) \end{aligned}$$

Combining these equations, we get

$$\begin{aligned} P(\hat{y}|x) &= \frac{P(x|\hat{y})P(\hat{y})}{P(x|\hat{y})P(\hat{y}) + P(x|\neg\hat{y})P(\neg\hat{y})} \\ &= \frac{\alpha \cdot P(x|y)P(y)}{\alpha \cdot P(x|y)P(y) + \beta \cdot P(x|\neg y)P(\neg y)} \\ &= \frac{\alpha a}{\alpha a + \beta b} \end{aligned}$$

Given that $b = \frac{a(1-p)}{p}$ (with $p = P(y|x)$), we can rewrite this as:

$$\begin{aligned} P(\hat{y}|x) &= \frac{\alpha a}{\alpha a + \beta \frac{a(1-p)}{p}} \\ &= \frac{\alpha p}{\alpha p + \beta(1-p)} \end{aligned}$$

When using the values as seen in Table 3.1, we get the following values for α and β.

$$\begin{aligned} pos = \alpha &= \frac{0.013}{0.69} \\ neg = \beta &= \frac{0.987}{0.31} \end{aligned}$$

Resulting in the final transformation function

$$\hat{p} = f(p) = \frac{\frac{0.013}{0.69} \cdot p}{\frac{0.013}{0.69} \cdot p + \frac{0.987}{0.31} \cdot (1 - p)} \tag{3.1}$$

with $f(p)$ converting p, which is the probability of the model on the training set, to a probability corresponding to the distribution of the real-life set ($\hat{p}$).

	Positive	Negative
Train	69%	31%
Real-life	1.3%	98.7%

Table 3.1: *Percentages of positive and negative binding data of downloaded (Train) data set and real-life data.*

To test this conversion function, which is created analogously to the conversion function found in [15], we try it for 3 different cases: The models prediction is very sure of its classification, either positively or negatively, or very unsure of its classification and it makes a random guess.

case	predicted probability	converted probability
certainly positive	99.0%	36.94%
certainly negative	5.0%	0.03%
random guess	69.0%	1.30%

Table 3.2: *To test the conversion function, we simulate three cases: the models prediction is highly certain of a positive result (predicted probability of binding is 99%), the models prediction is highly certain of a failed binding (predicts 5% binding probability) or the model takes a random guess (given the data distribution, this means a probability of 69%). We can clearly see that the conversion function shifts these probabilities to the wanted distribution (where a 'wild guess' would have a 1.3% chance of being positive, corresponding to the actual chance of a positive result when randomly selecting a HLA-peptide pair).*

Chapter 4

Evaluation

4.1 Results

This section will list and discuss all the results generated throughout this research.

4.1.1 Activation functions

This subsection lists the results obtained when comparing different activation functions. All tests were performed on a dataset of size 50.000, storing the best results after 5 epochs. In reality we use a larger dataset and more epochs, but since the interest was only in comparing different functions this faster training setup was used.

function	accuracy (%)	precision (%)	recall (%)
softmax	56.68	56.68	**100.00**
tanh	60.43	59.09	98.11
linear	59.29	57.65	99.53
sigmoid	**80.75**	**78.11**	91.75
hard sigmoid	57.49	57.20	99.29

Table 4.1: *A comparison of activation functions in the output layer for binary classification. The sigmoid function has better scores for accuracy and precision. Softmax performs better on recall. Since softmax scores 100% on recall, it is safe to assume that it simply* ***predicts*** True *for all entries. This is not very useful.*

function	accuracy (%)	precision (%)	recall (%)
prelu	60.70	60.70	**100.00**
leakyrelu	**64.38**	**65.12**	88.98
relu	61.20	61.20	98.62
tanh	64.21	64.52	81.18
hard sigmoid	60.70	60.70	**100.00**
linear	39.97	54.34	68.87

Table 4.2: *This table shows a comparison of activation functions used in the hidden layers of the neural network. LeakyRule scores noticeably better than the other functions when considering accuracy and precision. PRelu and Hard Sigmoid score best for recall. Similarly to Table 4.1 this is probably because these functions return* True *for all entries.*

These results show us that LeakyRelu is the best choice for the activation function in the hidden layers and sigmoid is the best choice for the classification layer.

4.1.2 Architectures

This subsection lists the results of different architectures. As opposed to the previous subsection, these results were generated by using the entire dataset unless explicitly stated otherwise, this amounts to approx. 200k elements. We compare the different architectures with eachother at different times and show the evolution of the performance of the architectures

as the project evolved. For earlier models we only have binary accuracy as metric, whereas the more informative metrics *precision* and *recall* were added for analysing the performance of later models.
As stated in Chapter 3, Subsection 3.5.2 we started with a basic model that was used as the foundation for further experimentation. This basic model uses LSTM layers as recurrent layers since these performed best for our problem (See Table 4.3).

type	accuracy (%)
SimpleRNN	48.95
GRU	68.98
LSTM	**70.17**

Table 4.3: *Comparison of different types of RNN layers. We find LSTM to perform best for our specific case.*

Since sequences of aminoacids don't have any directionality, we wanted to cover all grounds by making the recurrent layers bidirectional. We found that this almost doubled the training time, but actually worsened the performance on the testset (Table 4.4).

type	accuracy (%)	precision (%)	recall (%)
Regular LSTM	**71.12**	**81.96**	**61.73**
Bidirectional LSTM	67.01	75.71	60.01

Table 4.4: ***Bidirectional LSTM test*** *Making the LSTM layers bidirectional actually decreased performance of the model.*

The first adaptation we made to our basic model was to add Dropout layers with a dropout rate of 50%. The rate was chosen this high because the model suffered of a lot of overfitting (The network achieved an accuracy of over 20% higher on the training set than on the test set). The resulting architecture can be seen in Figure 4.1.

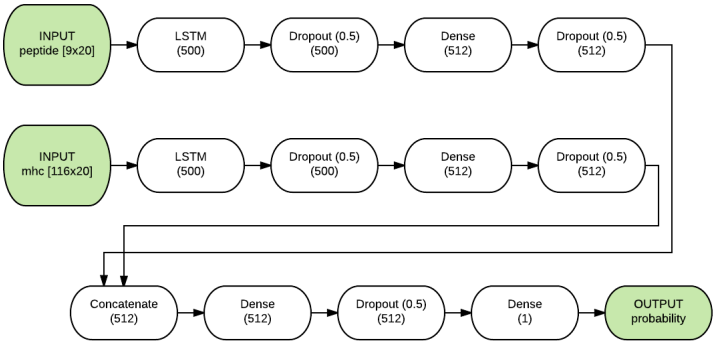


Figure 4.1: ***WS version 1*** *The first design for a neural network with Dropout layers.*

The performance of this model wasn't very good. In an attempt to fix this a deeper and more narrow architecture was tried. Since this network performed even worse than the original, we lowered the dropout rates to 20% in an attempt to improve the results by retaining more information. The resulting architecture can be seen in Figure 4.2.

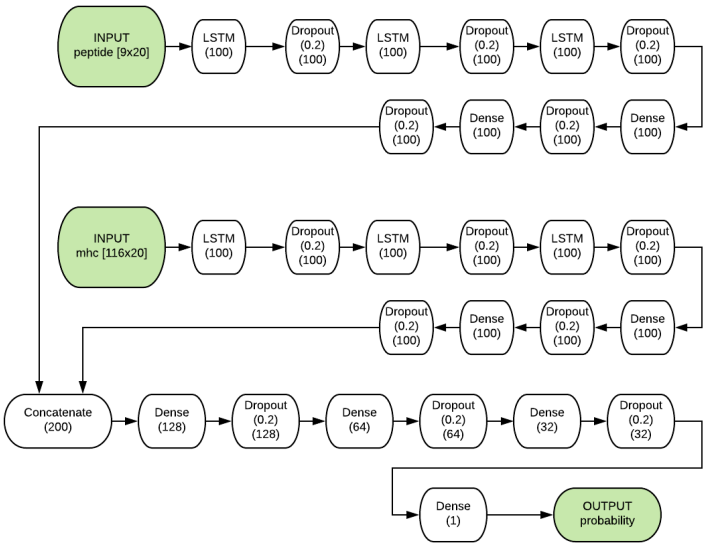


Figure 4.2: ***DN version 1*** *The updated design for a neural network with Dropout layers. It is deeper than the original, this should enable it to learn higher level features, improving performance.*

The reduction in dropout rate improved the results somewhat, but unfortunately it still performed similarly on the test set: It still showed signs of overfitting, even with the dropout rates of 20%.

The performance of a deeper neural network was better, but not as good as expected. Nonetheless the concept of a deeper neural network still appealed to us. We decided to proceed with both architectures with the goal to improve both the WS and the DN implementations.

In an attempt to further increase accuracy on the test data a wider architecture was tried. The width of the first layer of the WS network was doubled from 500 to 1000, at the same time the dropout rate was lower from 50% to 20%. This significantly increased testing accuracy. In an attempt to also improve the performance of the DN version it was widened a lot (from 100 to 1000). Since this made the training of the network unfeasible (approx. 25 times slower), we removed all but the first LSTM layers. These architectures can be seen in Figures 4.3 and 4.4 resp.

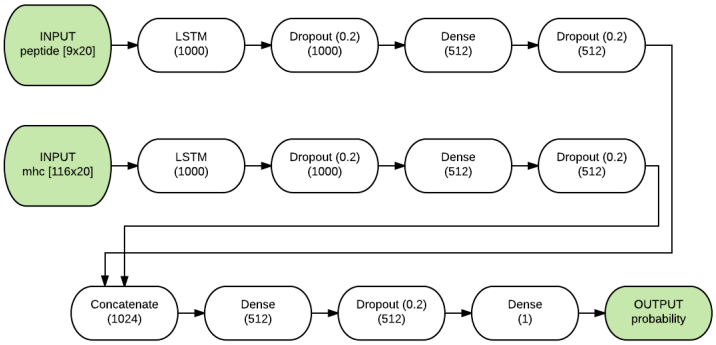


Figure 4.3: ***WS version 2***

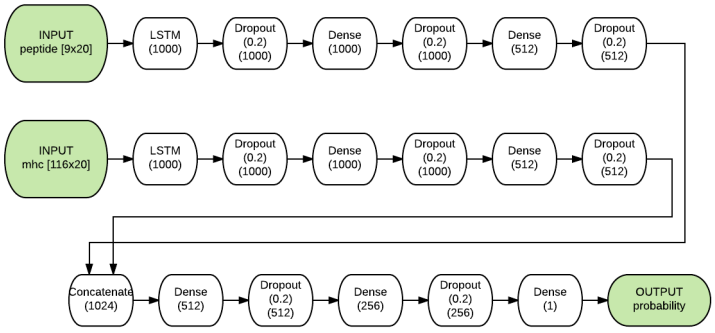


Figure 4.4: ***DN version 2***

The results of these four architectures are shown in Table 4.5. This table shows that in the first version, the DN architecture slightly outperformed

the WS architecture. When comparing the second versions, the WS architecture performs significantly better than the DN version. This tells us that a dropout rate that is set too high (i.e. 50%) can decrease performance of the model. It is apparent that the dropout rate is something that has to be fine-tuned more precisely. At the same time we see a great improvement when widening the WS architecture, unfortunately this same improvement does not happen when widening the DN architecture.

type	accuracy (%) (version 1)	accuracy (%) (version 2)
Wide and Shallow	63.48	**78.21**
Deep and Narrow	**65.62**	69.75

Table 4.5: ***Comparison of WS and DN versions 1 and 2*** *This table shows the accuracies achieved when testing the respective architectures on unseen proteins. Increasing the width of the WS network had a very impressive impact. The increase in performance was not so significant for the DN architecture.*

Even with the improvements, both architectures still suffer from overfitting. To finally improve this overfitting on an architectural level, we decided to again lower the width of the networks since a larger amount of features tends to overfit the network.
As a second fix we again increased the dropout between each layer. Since an increase in width showed an increase in accuracy, the number of features in the first layer were deemed too important to drop such a high percentage of them. For this reason we did not raise dropout on the first (LSTM) layers. Experimentation showed that in fact using dropouts on that first layer had a negative effect (see Table 4.6 for the comparative results) on the performance. For this reason we kept the dropout on the first layer to a rather low value of 10% (see Figure 4.5).

The width of this model was again reduced since further experimentation showed that a width of approx. 500 performed better than a width of 1000 for the current setup. These changes improved performance quite nicely.

type	accuracy (%)	precision (%)	recall (%)
40% dropout on LSTM	67.85	55.62	68.93
10% dropout on LSTM	75.24	62.76	**82.93**
0% dropout on LSTM	**75.80**	**69.50**	62.75

Table 4.6: ***LSTM dropout comparison*** *Results of the WS network with high dropout (40%) on the LSTM layers vs low dropout (10%) on the LSTM layers vs no dropout (0%) on the LSTM layers. When purely focussing on the precision, we would have to remove any dropout on the LSTM layer. However, we deemed the increase in recall when using 10% to outweigh the smaller improvement on precision when using 0%. For this reason we proceeded to use 10% dropout on the LSTM layers.*

In an attempt to improve the DN version of the network, we drastically reduced the width from 1000 to 256. This was again combined with a selective increase in dropout on each layer. As an extra we increased the depth of the network even further. The resulting architecture is shown in Figure 4.6.

The results achieved with these architectures can be seen in Table 4.9. These were the best results we achieved on an architectural level, meaning we had found our final architectures.

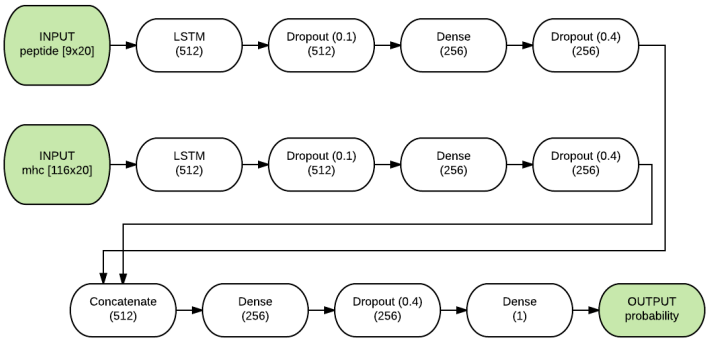


Figure 4.5: ***Final WS architecture***

After having found our favorite architectures, experiments were done to determine the optimal aminoacid encoding technique. As discussed

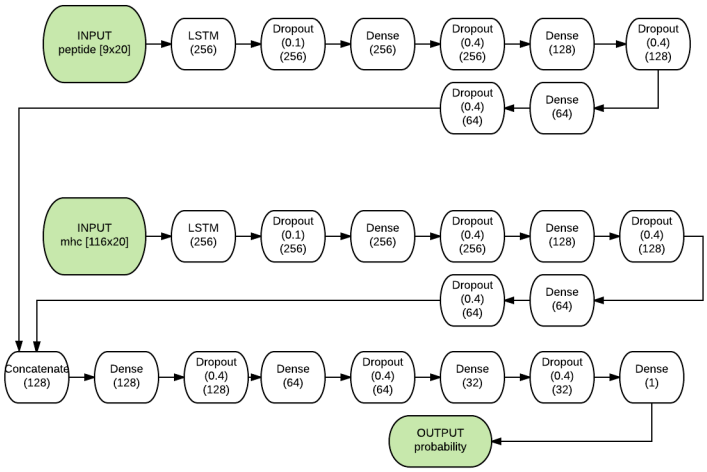


Figure 4.6: ***Final DN architecture***

in Section 3.4 we have two interesting options to explore: Using a BLOSUM90 substitution matrix and/or using the physicochemical properties of the aminoacids. The results in Table 4.7 clearly show that using physicochemical properties is not the best strategy for this problem.

encoding technique	accuracy (%)	precision (%)	recall (%)
BLOSUM90	**82.30**	**45.74**	36.06
PhysChem	30.31	17.85	88.91
Combination	40.53	38.85	**91.48**

Table 4.7: *A comparison of BLOSUM90 matrix as encoding technique vs using the physicochemical properties of an aminoacid vs using both simultaneous.*

Another technique tried in order to improve performance was to artificially augment the data. When creating random sequences for both MHC and peptide sequences the chances of having a positive reaction between them is so small that we can discard the possibility. We created 300.000 pairs of random peptide (lengths 9 or 10) and MHC (length 116) sequences. The results shown in Table 4.8 show that this was not a useful technique. We believe the reason for this is that the randomly generated sequences are too random to be of use for training.

type	accuracy (%)	precision (%)	recall (%)
without data augmentation	**75.24**	62.76	**88.93**
with data augmentation (+300 k)	73.21	**65.75**	59.00

Table 4.8: *Effects of data augmentation on predictive capacity. After adding 300.000 extra elements, we see a slight decrease in overal performance.*

As described in the previous chapter, eventually we decided to add a simple version of the model. This model contains one LSTM layer per input, one Dense layer on the concatenated featurevector and one classification layer. The resulting architecture (SM architecture) can be found in Figure 4.7. Table 4.9 shows the performance of these three final mod-

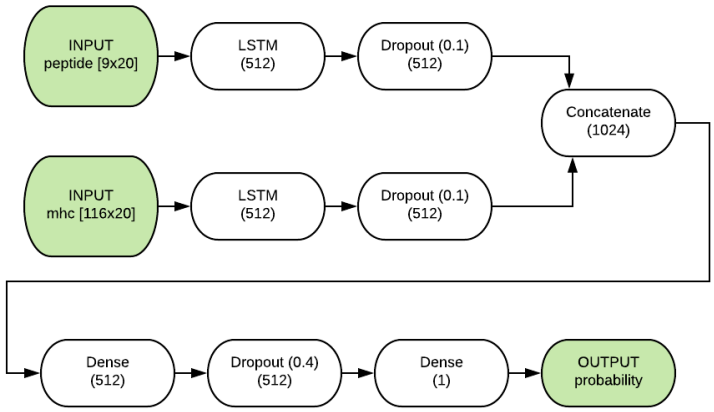


Figure 4.7: ***SM architecture***

els. The Simple model outperforms the other two, it does this however with quite a small margin. It appears that the exact architecture is not the most important issue for this specific problem, other parameters such as the dropout rate and activation functions have a more substantial effect.

type	accuracy (%)	precision (%)	recall (%)
Wide and Shallow	84.73	84.94	**97.89**
Deep and Narrow	83.50	85.23	95.54
Simple	**86.55**	**88.16**	95.72

Table 4.9: *General architecture results: comparison done with optimal parameters and the standard logloss cost function.*

After having added the weighted log loss function (which makes negative results twice as expensive as positive results), we reran the training and testing of the three models (See Table 4.10). The effect is noticed by the shift from a higher recall score in the original version to a higher precision score in the new version. The higher cost for negative results means that false positives (FP) are punished more severely than false negatives (FN), therefore the number of FP will decrease and the number of FN will increase. If we consider the formulas for precision and recall (Equations 4.1 and 4.2 respectively), it is clear why this would lead to a higher precision and lower recall:

$$precision = \frac{TP}{TP + FP} \tag{4.1}$$

$$recall = \frac{TP}{TP + FN} \tag{4.2}$$

The changes in scores moved the performance of the Simple model from the first place to the third place. The difference between performance of these models is so small however, that we consider this to be irrelevant.

	accuracy (%)	precision (%)	recall (%)
Wide and Shallow	**85.24**	92.97	**87.84**
Deep and Narrow	85.10	**94.43**	86.10
Simple	85.04	93.02	87.51

Table 4.10: *Different architectures and their results after implementing the weighted log loss function.*

4.1.3 General model evaluation

Unseen proteins

All results in Subsection 4.1.2 are computed by training the network on a set of proteins, validated on another set of proteins and eventually tested on a final set of proteins. These three sets are created in such a way that no protein is found in more than one set, regardless of how many experimental data with this protein is available. For this reason we refer to Table 4.10 for the results achieved when testing on unseen proteins.

Unseen protein family

	accuracy (%)	precision (%)	recall (%)
Wide and Shallow	**51.06**	**93.07**	**40.27**
Deep and Narrow	37.18	89.16	22.18
Simple	44.81	86.17	34.86

Table 4.11: *Results of testing on unseen families of HLA-A, HLA-B and HLA-C. The tested families (*HLA-A*23, HLA-B*83, HLA-C*04, HLA-C*07, HLA-B*49, HLA-A*29*) were selected randomly*

4.1.4 Pretraining

The idea behind pretraining is that a model is pretrained on related but different data than what the model will be tested on. After this pretraining, the model is retrained on a (possibly very small) set of data similar to the test data. For this thesis we did a division on peptide length (length of 9 and length of 10) and on different HLA classes (HLA-A, HLA-B and HLA-C).

Pretrain on length 9

	accuracy (%)	precision (%)	recall (%)
pretrain length 9	76.60	74.85	88.44
pretrain length 9, retrain length 10	**81.85**	**80.94**	89.15
train length 10	56.82	56.87	**98.58**

Table 4.12: *Using pretraining on different lengths looks very promising. We achieve better performance when pretraining the model on length 9 and testing it on length 10 than when only training and testing on length 10. Knowing this, it is not surprising that pretraining on length 9 and retraining on length 10 yields the best results.*

Pretrain on HLA-A and HLA-B

After the succesful test using pretraining on different peptide lengths, we redid the experiment using different HLA classes. We test on HLA-C because it is the smallest class and could benefit most of using pretraining.

	accuracy (%)	precision (%)	recall (%)
Wide and Shallow	**81.09**	81.34	**99.62**
Deep and Narrow	18.62	0.00	0.00
Simple	69.38	**81.65**	80.45

Table 4.13: *Pretrain the networks on HLA-A and HLA-B only then test them on HLA-C. The WS architecture clearly performs better in this case.*

	accuracy (%)	precision (%)	recall (%)
Wide and Shallow	89.00	**95.09**	93.19
Deep and Narrow	**89.30**	94.94	**93.69**
Simple	85.39	94.89	89.38

Table 4.14: *Pretrain the networks on HLA-A and HLA-B, then retrain the networks on (a subset of) HLA-C. Eventually test the networks on HLA-C. The difference in performance between the three networks is significantly smaller.*

	accuracy (%)	precision (%)	recall (%)
Wide and Shallow	90.01	94.86	94.57
Deep and Narrow	**93.38**	94.88	**98.31**
Simple	89.24	**95.05**	93.50

Table 4.15: *Train the networks only on (a subset of) HLA-C, then test on the rest of HLA-C.*

4.1.5 Real life model comparison

At the final stages of the research for this thesis we received a dataset constructed from experiments performed on the blood of patients. The extracted MHC and peptide pairs were fed to a model that predicts the binding affinity between the sequences. The distribution of these values can be seen in Figure 4.8.

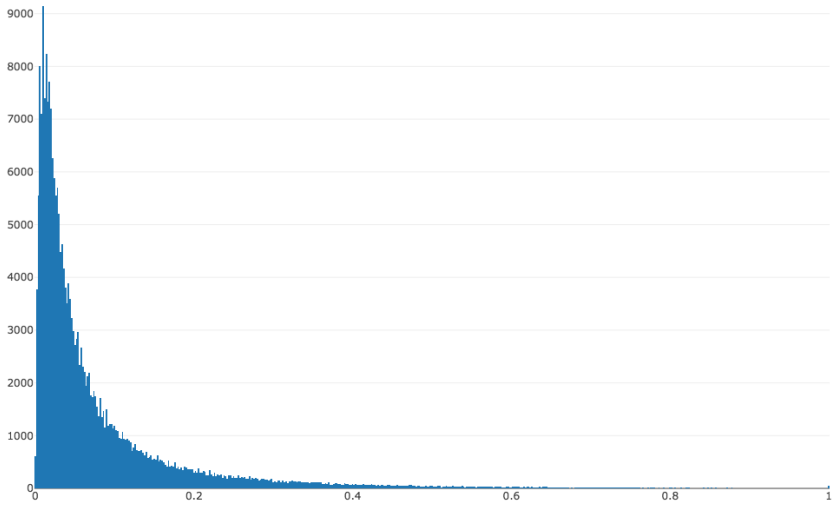


Figure 4.8: *Distribution of the predicted affinity of hospital patients.*

The dataset contains the MHC name, the peptide sequence and some (predicted) numeric values. The first 10 elements of this set are shown in Listing 4.1 where the column *mhc_norm* corresponds to the normalized affinity.

	comb	comb_norm	hla	mhc	mhc_norm	peptide	protein
0	0.35868	0.414018	HLA–A01:01	0.137	0.205793	MASTTPITM	sp|P19725|POLS_
1	0.01770	0.078030	HLA–A01:01	0.033	0.047256	ASTTPITME	sp|P19725|POLS_
2	0.11500	0.173906	HLA–A01:01	0.089	0.132622	STTPITMED	sp|P19725|POLS_
3	0.32773	0.383521	HLA–A01:01	0.090	0.134146	TTPITMEDL	sp|P19725|POLS_
4	0.01171	0.072128	HLA–A01:01	0.012	0.015244	TPITMEDLQ	sp|P19725|POLS_
5	0.17834	0.236318	HLA–A01:01	0.030	0.042683	PITMEDLQK	sp|P19725|POLS_
6	0.19986	0.257523	HLA–A01:01	0.112	0.167683	ITMEDLQKA	sp|P19725|POLS_
7	0.30406	0.360197	HLA–A01:01	0.066	0.097561	TMEDLQKAL	sp|P19725|POLS_
8	−0.00598	0.054697	HLA–A01:01	0.034	0.048780	MEDLQKALE	sp|P19725|POLS_
9	−0.00330	0.057338	HLA–A01:01	0.013	0.016768	EDLQKALET	sp|P19725|POLS_

Listing 4.1: *First 10 elements of patient data*

We extracted the *mhc_norm* value of each row and converted it to binary. We did this conversion by considering all mhc_norm values larger than 0.5 to be positive and all those below 0.5 are considered negative. This gave a distribution of 3,670 positives elements and 232,364 negative elements, meaning that approx. 98.7% of the entries are negative and 1.3% of the elements are considered positive. The results of this experiment can be seen in Table 4.16. Overall we conclude that the WS version is the most useful version of the three when the goal is to make a useful generic model.

	accuracy (%)	precision (%)	recall (%)
Wide and Shallow	**97.38**	**4.51**	3.41
Deep and Narrow	83.32	3.38	35.26
Simple	65.33	2.97	**67.30**

Table 4.16: ***Patient data prediction*** *Our predicted results are very different from the classifications we received.*

4.2 Discussion

After having experimtented with different architectures, parameters and datasets, this section will try to encapsulate the most important findings that were discovered during this research.

During our research we find that predicting the bindingpreferences of an

MHC molecule with a given peptide is very doable with current techniques. Creating a more generic model which ideally performs good on unseen molecules is unfortunately much more challenging. We found that an optimal architecture with finely-tuned parameters is crucial to achieve success.

Architecture As mentioned in Subsection 3.5.2, we implemented a LSTM layer as recurrent layer because both LSTM and GRU performed much better than SimpleRNN. LSTM and GRU both solve the Vanishing Gradient problem, a difficulty which SimpleRNN can't handle. This shows that there is a relation between elements that are far away from eachother in the sequence.

Contrary to what we assumed making the LSTM layers bidirectional actually decreased performance. We presume that this happens because making LSTM bidirectional doubles the number of parameters to be trained. This increase is so significant that the models problem of overfitting increases in such a way that the performance is affected in a negative way.

We found that an increase in dropout in the first (LSTM) layer resulted in a decrease in performance of the model. We believe this happens because the network needs to learn many features in order to be able to make an accurate prediction. A high amount of dropout at the beginning results in a loss of too much information, which the network can't recover from.

Similarly we noticed that an increase in width in the *WS* network resulted in an increase in performance (accuracy rose with approx. 15%). This strengthens our believe that allowing the network to learn a large number of features is crucial for succesfully predicting the MHC-peptide binding preferences.

In contrast we noticed that an increase in width in the DN network only

resulted in a minor improvement in performance. We believe this is the case because such a large increase in width, on a network that is already very deep, increases the complexity too much. We believe this increases the amount of overfitting as well as the performance, resulting in a small increase in overal performance (accuracy rose with approx. 4%).

Datasets During the research and writing of this thesis we were confronted with the importance of *intelligently* dividing the dataset at hand into *three subsets*:

- A training set
- A test set
- A validation set

The training set is used to train the data, for this reason it should be the largest of the three sets. The test set is used to test the performance of the model, it can be relatively small but should still be representative of the entire dataset in order to give a realistic view of the performance of the model. The validation set is used during training to validate the performance of the model. It is separated from the other two sets so to not overfit the data too much.
During our research we noticed that the model would sometimes perform worse after more epochs. This happened because the model wasn't validated during training. To solve this the models performance is measured against the validation set after each epoch. Only if the performance of the model on the validation set improved, then the model is stored. At the end of the training process, the last stored model is used. This doesn't necessarily correspond to the result of the last epoch.
As mentioned above it is not only important to divide the dataset into three subsets. It is also very important to do it intelligently. At first we divided the dataset randomly. As explained in Subsection this resulted in a training set that contained entries of all proteins that were also found in the test set. To solve this we had to divide the dataset based on the proteins

of the entries, not just the entries themselves. As a bonus, we also added the possibility to divide the dataset based on the MHC-I class (HLA-A, HLA-B or HLA-C) or based on the protein families.

Metrics The default metric for a classification problem is *accuracy*, however this proved to be insufficiently informative for this problem, mainly due to the large discrepancy between negative and positive samples. More detailed metrics such as precision and recall are much more informative on the performance of the model. The confusion matrix especially is a highly informative tool to analyse the performance of a classification model. In certain cases of highly imbalanced classes a model might perform very well when only taking accuracy into account, when in reality if fails completely. This is illustrated in the fictive example in Table 4.17.

1000	5
15	0

Table 4.17: *This confusion matrix shows a poorly performing classifier. When only regarding the accuracy however, we get an impressive result:* $\frac{1000+0}{1000+5+15+0} = 98\%$*. For more information we also calculate precision (*$\frac{0}{5+0} = 0\%$*) and recall (*$\frac{0}{15+0} = 0\%$*). This shows that the model doesn't perform as good as one would think based on the accuracy alone.*

Handle imbalanced data It is often the case that an imbalance exists in the datasets. This can be an imbalance between classes, an imbalance in distribution between training and test sets or an implicit imbalance created by a difference in importance of one class over another. Unfortunately we encountered all three cases of imbalance in this thesis: The downloaded datasets we used for training and testing were relatively well balanced. We were fortunate that we had quite a lot of data, this meant that the balance between training and test set remained more or less the same after dividing them.
The largest difficulty when regarding these sets was the higher cost for false positives than for false negatives. We solved this problem by intro-

ducing a weighted log loss cost function (as dicussed in Subsection 3.5.3). This cost function penalizes false positives twice as high as false negatives. The bigger problem arised when we received the actual patient dataset at the end of our research. This dataset was highly biased towards the negative side (98.7%). Since the patient data was never used for any training, techniques to handle this bias (such as over- or undersampling) were not necessary. The more important problem was the difference in distrubtion between the patient data and the models training data. When a general model doesn't know if it has to classify something as 1 or 0, it should return a value of 0.5. On a dataset where 98.7% is negative and only 1.3% is positive, the average 'guess' of the model should correspond to 0.013. We solved this by creating a conversion function, also explained in Subsection 3.5.3.

Chapter 5

Conclusion

In this thesis we explored the possibilities of creating a deep learning model using recurrent neural networks to predict the binding preferences of MHC proteins.

Different architectures perform quite similarly when testing on unseen proteins. The true difference in architecture arises when we test on unseen MHC protein classes. In this case a wide network seems to perform better than a narrow network, implying that learning a large number of features is key. It might be useful to further explore the possibilities of predicting the binding preferences of TCR with peptides. Since very little data is available on TCR binding preferences, it might be possible to use a model that was pretrained on MHC-peptide pairs. We have already shown that a model can predict binding preferences on unseen HLA classes. In addition the performance increased greatly when retraining the model on a small amount of the tested HLA class. Possibly pretraining a model on MHC-peptide pairs and retraining it on the small number of available TCR-peptide pairs will result in a very useful model.

The different architectures were tested on patient data with a very different distribution than the training data. An important note however is that we assumed the predictions from the unknown model to be correct,

which is not necessarily true. When we compare these classifications with our own predictions we get very different results, even with the added techniques to mitigate this imbalance. We do get a small intersection of positive results between both predictive models. Future experiments will have to determine if this also corresponds to a higher success rate.

Bibliography

[1] Morten Nielsen, Claus Lundegaard, Thomas Blicher, Kasper Lamberth, Mikkel Harndahl, Sune Justesen, Gustav Røder, Bjoern Peters, Alessandro Sette, Ole Lund, and Søren Buus. NetMHCpan, a Method for Quantitative Predictions of Peptide Binding to Any HLA-A and -B Locus Protein of Known Sequence. *PLoS ONE*, 2(8):e796, August 2007.

[2] Hao Zhang, Ole Lund, and Morten Nielsen. The PickPocket method for predicting binding specificities for receptors based on receptor pocket similarities: application to MHC-peptide binding. *Bioinformatics*, 25(10):1293–1299, May 2009.

[3] Laurent Jacob and Jean-Philippe Vert. Efficient peptide-MHC-I binding prediction for alleles with few known binders. *Bioinformatics (Oxford, England)*, 24(3):358–366, February 2008.

[4] Nebojsa Jojic, Manuel Reyes-Gomez, David Heckerman, Carl Kadie, and Ora Schueler-Furman. Learning MHC I–peptide binding. *Bioinformatics (Oxford, England)*, 22(14):e227–235, July 2006.

[5] Cytotoxic t cell. `https://en.wikipedia.org/wiki/Cytotoxic_T_cell`.

[6] The twenty amino acids. `http://www.cryst.bbk.ac.uk/education/AminoAcid/the_twenty.html`.

[7] Warren S. Sarle. Neural networks and statistical models, 1994.

[8] Tushar Gupta. Deep learning: Feedforward neural network.

[9] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[10] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. In *Deep Learning Workshop, International Conference on Machine Learning (ICML)*, 2015.

[11] Eamonn Keogh and Abdullah Mueen. *Curse of Dimensionality*, pages 257–258. Springer US, Boston, MA, 2010.

[12] Christopher Olah. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015.

[13] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting, 2014.

[15] Mike Swarbrick Jones. Cross entropy and training-test class imbalance. `https://swarbrickjones.wordpress.com/2017/03/28/cross-entropy-and-training-test-class-imbalance/`, 2017.

[16] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[17] Chandirasegaran Massilamany, Sally A Huber, Madeleine W Cunningham, and Jay Reddy. Relevance of molecular mimicry in the mediation of infectious myocarditis. 7, 11 2013.

[18] Rohit Bhattacharya, Ashok Sivakumar, Collin Tokheim, Violeta Beleva Guthrie, Valsamo Anagnostou, Victor E. Velculescu, and Rachel

Karchin. Evaluation of machine learning methods to predict peptide binding to MHC Class I proteins. July 2017.

[19] Lianming Zhang, Keiko Udaka, Hiroshi Mamitsuka, and Shanfeng Zhu. Toward more accurate pan-specific MHC-peptide binding prediction: a review of current methods and tools. *Briefings in Bioinformatics*, 13(3):350–364, May 2012.

[20] Pavel P. Kuksa, Martin Renqiang Min, Rishabh Dugar, and Mark Gerstein. High-order neural networks and kernel methods for peptide-MHC binding prediction. *Bioinformatics (Oxford, England)*, 31(22):3600–3607, November 2015.

[21] Michael Rasmussen, Emilio Fenoy, Mikkel Harndahl, Anne Bregnballe Kristensen, Ida Kallehauge Nielsen, Morten Nielsen, and Søren Buus. Pan-Specific Prediction of Peptide-MHC Class I Complex Stability, a Correlate of T Cell Immunogenicity. *Journal of Immunology (Baltimore, Md.: 1950)*, 197(4):1517–1524, August 2016.

[22] Heng Luo, Hao Ye, Hui Wen Ng, Sugunadevi Sakkiah, Donna L. Mendrick, and Huixiao Hong. sNebula, a network-based algorithm to predict binding between human leukocyte antigens and peptides. *Scientific Reports*, 6(1), October 2016.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436, May 2015.

[24] Seonwoo Min, Byunghan Lee, and Sungroh Yoon. Deep learning in bioinformatics. *Briefings in Bioinformatics*, 18(5):851–869, September 2017.

[25] James Robinson, Jason A. Halliwell, Hamish McWilliam, Rodrigo Lopez, Peter Parham, and Steven G. E. Marsh. The IMGT/HLA database. *Nucleic Acids Research*, 41(D1):D1222–D1227, October 2012.

[26] Neha Yadav, Anupam Yadav, and Manoj Kumar. History of Neural Networks. In *An Introduction to Neural Network Methods for Differential Equations*, pages 13–15. Springer Netherlands, Dordrecht, 2015. DOI: 10.1007/978-94-017-9816-7_2.

[27] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.