CNN-PepPred: User's Guide

Valentin Junet[*,†] Xavier Daura[†,‡,§]

October, 2021
Version 0.1.1

Contents

[*]Anaxomics Biotech SL, Barcelona 08008, Spain
[†]Institute of Biotechnology and Biomedicine, Universitat Autònoma de Barcelona, Cerdanyola del Vallès 08193, Spain
[‡]Catalan Institution for Research and Advanced Studies (ICREA), Barcelona 08010, Spain
[§]Biomedical Research Networking Center in Bioengineering, Biomaterials and Nanomedicine (CIBER-BBN), Spain

1 Installation

1.1 Installation with *conda*

In the folder *CNN-PepPred*, you will find environment files to create a python environment with all the required packages to run the model. There are two environment files, *model_environment_gpu.yml* and *model_environment_cpu.yml*, the first one will create an environment to work with GPUs and the second one with CPUs. GPU computations will usually be faster than CPU ones. The environment contains the following packages:

- python version 3.6.10
- numpy version 1.19.1
- tensorflow-gpu or tensorflow (for CPU environment) version 2.0.0
- keras-gpu or keras (for CPU environment) version 2.3.1
- pandas version 1.1.3
- pathlib
- biopython version 1.78
- logomaker
- scikit-learn version 0.23.2
- seaborn version 0.11.0
- pillow version 8.0.0

To create the environment, set the working directory to be the folder *CNN-PepPred* and type the following in your Anaconda terminal:

```
conda env create -f model_environment_gpu.yml
```

for the GPU environment and

```
conda env create -f model_environment_cpu.yml
```

for the CPU environment. This might take a few minutes.
Once the installation is finished, activate the environment using the command

```
conda activate CNNPepPred_Env_GPU
```

for GPU and

```
conda activate CNNPepPred_Env_CPU
```

for CPU.
At the end of the session, you can deactivate the environment using the command

```
conda deactivate
```

To remove the environment, use the command

```
conda remove --name CNNPepPred_Env_GPU --all
conda remove --name CNNPepPred_Env_CPU --all
```

1.2 Installation with *pip*

It is recommended to use the *conda* installation since the 3.6 version of python is required and creating an environment in Anaconda is more convenient and more uniform through different operating systems. However if you wish to do the installation using *pip*, make sure that you are using python 3.6 and create an environment and install the required packages following the instructions below.
Set the main folder *CNN-PepPred* as working directory and create a python environment called *CNNPepPred_Env_GPU* or *CNNPepPred_Env_CPU* using the lines

```
python -m venv CNNPepPred_Env_GPU
python -m venv CNNPepPred_Env_CPU
```

Activate the environment on Linux or MacOS with

```
source CNNPepPred_Env_GPU/bin/activate
source CNNPepPred_Env_CPU/bin/activate
```

and on Windows with

```
.\CNNPepPred_Env_GPU\Scripts\activate
.\CNNPepPred_Env_CPU\Scripts\activate
```

To install the required packages, as listed in the previous subsection use, for the GPU environment

```
pip install -r requirements_GPU.txt
```

and for the CPU environment

```
pip install -r requirements_CPU.txt
```

To deactive the environment, run

```
deactivate
```

1.3 Test

To test the installation, call the main function with the template *test_template.txt* in the *Test* folder. It will apply a pre-trained model to the sequences *test_seq.fasta*. The template contains pathways to the pre-trained model and to the data; you will need to modify these pathways in the template to be adapted to the operating system of your computer and replace *[your_working_path]* by the pathway of the folder *CNN-PepPred*. The result file *HLA_DRB1_08_01_predictedOutcome.txt* will be saved in the same folder. Check that they match the results in the file *HLA_DRB1_08_01_predictedOutcome_to_obtain.txt*.
To apply the main script, activate the previously installed environment and set the working directory to be the folder *CNN-PepPred*. If you are working from the python console, execute the lines

```
import sys
model_from_template = open("model_from_template.py").read()
sys.argv = ['model_from_template.py','test_template.txt']
exec(model_from_template)
```

Alternatively, you can run

```
import model_from_template
modelCNN = model_from_template.main('test_template.txt')
```

Or, if you are working from Spyder, you can execute the line

```
runfile('model_from_template.py',args='test_template.txt')
```

2 Description

The main folder *CNN-PepPred* contains two python scripts, *model_initializer.py* and *model_from_template.py*. The first contains the class *CNNPepPred*, where all the functions for training and applying allele-specific models are defined, the second launches the analysis following a user-filled template.

2.1 The class *CNNPepPred*

The class *CNNPepPred* is in the python script *model_initializer.py* and contains the following methods.

2.1.1 *__init__*

Description
Initialize the class. The input arguments can be read from the template.

Usage

```
CNNPepPred(allele='no_allele_name',savePath=Path(os.getcwd()),
   doTraining=False,trainingData=None,trainingOutcome=None,
   doLogoSeq=False,doCV=False,cvPart=None,kFold=5,doApplyData=
   False,trainedModelsFile=None,applyData=None,applyDataName=
   None,epitopesLength=15,parametersFile='parameters.txt')
```

Arguments

`allele`

The name of the allele.

`savePath`

The pathway where to save the results.

`doTraining`

Whether or not to do the training.

`trainingData`

The training sequences, in a list.

`trainingOutcome`

The training outcome corresponding to the training sequences.

`doLogoSeq`

Whether or not to plot (logo plot) the core binding pattern of the trained model.

`doCV`

Whether or not to perform a cross-validation.

`kFold`

The number of fold for the cross-validation.

`doApplyData`

Whether or not to apply the trained model to new sequences.

`trainedModelsFile`

The file containing the trained model. This option is only valid if no training is selected. The file is a pickle saved file from a previous training using this class.

`applyData`

The new sequences for the application of the trained model.

`applyDataName`

The name of the new sequences.

`epitopesLength`

The length of the epitopes on which the trained model will be applied. Each new sequence will be cut into all overlapping *epitopesLength*-mers and a prediction will be made for each of them.

`parametersFile`

The name with extension of the file containing the parameters of the model.

2.1.2 *getParameters*

Description
Get the parameters of the model as given by the parameter file of the template. The parameters will be saved as attributes. For more information about the parameters, see section 2.2.

Usage

```
CNNPepPred.getParameters()
```

2.1.3 *aa2int*

Description
Transform a sequence of amino acids to integers according to:

A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	-
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

where "-" stands for the absence of amino acids. Any non-amino acid characters will be considered as "-".

Usage

```
CNNPepPred.aa2int(s)
```

Arguments

`s`

The amino-acid residue sequences in a list.

Value
Returns `sInt`, a list with the sequences as integers.

2.1.4 *int2aa*

Description
Transform a sequence of integers to amino acids according to the table in section 2.1.3.

Usage

```
CNNPepPred.int2aa(sInt)
```

Arguments

`sInt`

The integer sequences in a list of numpy arrays. If all sequences have the same length, it can be a numpy array of shape (N, L) where N is the number of sequences and L their length.

Value
Returns `s`, a list with the sequences as amino acid characters.

2.1.5 *seqLength*

Description
Compute the maximal length *maxL* in a set of sequences, the length *seqL* of each of them and the parameter *nMaxPool* determining the pooling size of the maxpooling layer in the model.
For more information on *nMaxPool*, see Appendix A.2.

Usage

```
CNNPepPred.seqLength(s,saveOutput=False)
```

Arguments

`s`

The sequences, which can be either a list of amino-acid residue sequences or a list of integer sequences.

`saveOutput`

Whether or not to save the outputs as attributes.

Value
Returns `seqL`, a numpy array with the length of the sequences, `maxL`, the maximal length and `nMaxPool`, the pooling size of the maxpooling layer.

2.1.6 *addEmptyPositions*

Description
Add the integer value 20, standing for the absence of amino acid, to the given sequences as needed so that they all have the same length, equal to the maximal length in the training set. In addition, it will add this value *nbPrev* times at the beginning of the sequences and *nbAfter* times at the end. The *nbPrev* and *nbAfter* parameters are set in the parameter file (section 2.2).
If the maximal length (the attribute *maxL* of the class) from a previously trained model is smaller than the maximal length in the training set (this can happen if the training is performed with transfer learning, see Appendix

A.5), the instances with length larger than *maxL* are removed from the training data.

Usage

```
CNNPepPred.addEmptyPositions(sInt)
```

Arguments

`sInt`

The integer sequences in a list.

Value
Returns `sIntNew`, a list of the integer sequences with the added absence-of-amino-acid values.

2.1.7 *getImages*

Description
Transform the sequences into images according to the given similarity matrix. For a given sequence, the height of the image corresponds to the residues of the sequence, the width corresponds to the 21 amino acids+absence of amino acids. The image is then filled with the similarity value between a residue of the sequence and an amino acid.
For more information on the peptide's encoding, see Appendix A.1.

Usage

```
CNNPepPred.getImages(sInt)
```

Arguments

`sInt`

The integer sequences in a numpy array as given by the output of *addEmptyPositions* (section 2.1.6). All sequences must have the same length.

Value
Returns `IM`, a 4D numpy array with the images corresponding to the sequences. The first dimension corresponds to the number of sequences, the second to the height, the third to the width and the fourth to the channel (which is always 1 with this encoding).

2.1.8 *trainCNN*

Description
Train an ensemble convolutional neural network model. The base model consists of a *Conv2D* layer with *ReLu* activation, a *MaxPooling2D* layer and a *Dense* (or fully connected) layer. The parameters are defined in the parameter file (section 2.2).
For more information on the model's architecture, see Appendix A.2.
If the class contains an attribute *trainedModels*, then these previously trained models will be used for transfer learning (see Appendix A.5).

Usage

```
CNNPepPred.trainCNN(IM,out,saveModel=False)
```

Arguments

`IM`

The training images, as given by the output of *getImages* (section 2.1.7).

`out`

The training outcome.

`saveModels`

Whether or not to save the trained model as an attribute and in the saving pathway *savePath* of the class. If *saveModels* is true, the computation time of the training will be an attribute of the class called *timeTrain*. A folder called *model_[allele]* (where [allele] is the allele name of the class) will be created, it will contain the parameter file of the model, the training data in a *txt* file and a folder called *nets* where the trained nets will be saved.

Value
Returns `models`, a list containing all the Keras trained models.

2.1.9 *applyCNN*

Description
Apply the trained model.

Usage

```
CNNPepPred.applyCNN(models,IM,saveOutcome=False)
```

Arguments

`models`

The ensemble model as given by the output of *trainCNN* (section 2.1.8).

`IM`

The images on which the trained model will be applied.

`saveOutcome`

Whether or not to save the predicted outcome as an attribute.

Value
Returns `yhat`, a numpy array with the predicted outcome of each sample.

2.1.10 *crossValCNN*

Description
Perform the training in a cross-validation set up. The computation time of the cross-validation will be an attribute of the class called *timeCV*.
If the training is performed with transfer learning (see Appendix A.5), the peptides containing shared l-mers (where l is the parameter determining the length of the core binder, 9 by default) with the pre-trained model used for transfer learning are removed from the test set but are kept for training.

Usage

```
CNNPepPred.crossValCNN(IM,out)
```

Arguments

`IM`

The training images for the cross-validation.

`out`

The training outcome.

Value
Returns `yhatCV`, a numpy array containing the cross-validated predicted outcome of each sample and `modelCV`, a list containing the trained Keras models (as returned by *trainCNN*, section 2.1.8) for each fold.

2.1.11 *feedForwardAndGetScore*

Description
Apply the trained model of the class to new sequences and get the score for each of the overlapping l-mers of a sequence, where l is the parameter determining the length of the core binder (9 by default).
To control the memory usage, the application of the sequences will be by batches of *maxNbSamples2apply*, which is a parameter (see section 2.2) with default value 50000.
For more information on the contribution score, see Appendix A.4.

Usage

```
CNNPepPred.feedForwardAndGetScore(seq,saveOutcome=False)
```

Arguments

`seq`

The sequences on which the trained model will be applied as given by the output of *addEmptyPositions* (section 2.1.6).

`saveOutcome`

Whether or not to save the predicted outcome as an attribute. If *saveOutcome* is true, the computation time to apply the model on the data will be an attribute of the class called *timeApply*.

Value
Returns `contributionScore`, a numpy array with the contribution score of all the overlapping l-mers of each sequence and `yhat`, a numpy array with the predicted outcome of each sequence.

2.1.12 *generateRandomSeq*

Description
Generate integer random sequences. The number of random sequences to generate is set in the parameter file (section 2.2).

Usage

```
CNNPepPred.generateRandomSeq(followLengthDistr=False)
```

Arguments

`followLengthDistr`

If `False`, all the random sequences will have the same length *lengthRandSeq* as given in the parameter file (section 2.2). If `True`, the length distribution of the random sequences will follow the length distribution of the training data saved as an attribute called *seqL* with the function *seqLength* (section 2.1.5)

Value
Returns `sR`, a list with the randomly generated integer sequences.

2.1.13 *plotLogoSeq*

Description
Generate a logo plot (using the package *logomaker*) of the highest scoring core binders. The plot will be saved in the pathway *savePath* of the class. The number of best scoring sequences used in the logo plot is set in the parameter file (section 2.2).

Usage

```
CNNPepPred.plotLogoSeq(contributionScore,yhatR)
```

Arguments

`contributionScore`

The contribution score of each overlapping l-mer in all of the sequences to which the trained model has been applied, as given by the output of *feedForwardAndGetScore* (section 2.1.11).

`yhatR`

The predicted score of each sequence.

Value
Returns `h`, the plot handle of the logo plot; `sBchar`, a list with the amino-acid sequences used to generate the plot and `pim`, the information matrix corresponding to the logo plot.

2.1.14 *computationTime*

Description
Save the computation time as an attribute called *timeTotal*.

Usage

```
CNNPepPred.computationTime(time_elapsed)
```

Arguments

`time_elapsed`

The elapsed time to save.

2.1.15 *getCVresults*

Description
Get the cross-validation results. The scores are: PC (Pearson correlation), AUC (area under the curve), RMSE (root mean square error), MCC (Matthews correlation coefficient), ACC (accuracy), BACC (balanced accuracy), F1 (F1-score). The result will be saved as a txt file *'cross_validation_results.txt* in the path *savePath*.

Usage

```
CNNPepPred.getCVresults()
```

2.1.16 *printApplyOutcome*

Description
Print the predicted outcome of the analysed sequences as a txt file *[allele]_predictedOutcome.txt* where [allele] is the allele name. The file will be saved in the path *savePath*. Note that only unique core binders will be printed; if there are different peptides with the same core, the one with the highest predicted outcome will be printed.

Usage

```
CNNPepPred.printApplyOutcome(saveTable = False)
```

Arguments

`saveTable`

Whether or not to save the output table as an attribute.

Value
Returns `table`, a pandas data frame with the predicted outcome of the sequences on which a trained model was applied. The table only contains unique core binders.

2.1.17 *seq2Lmer*

Description
Cut sequences into all overlapping *epitopesLength*-mers, where *epitopesLength* is as given in the template (section 2.3).

Usage

```
CNNPepPred.seq2Lmer(seq,nameSeq=None,takeUniqueLmer=True,
   saveLmer=False)
```

Arguments

`seq`

The integer amino-acid sequences in a list of numpy arrays.

`nameSeq`

The name of the sequences.

`takeUniqueLmer`

Whether or not to select only the unique overlapping *epitopesLength*-mers.

`saveLmer`

Whether or not to save the output sequences as an attribute.

Value
Returns `sLmer`, a list with all overlapping *epitopesLength*-mers as integers; `nameSeqLmer`, the name of the sequences each element of `sLmer` belongs to and `indLmer`, the indices of the sequences each element of `sLmer` belongs to.

2.1.18 *getCoreBinder*

Description
Get the core binders of the sequences.

Usage

```
CNNPepPred.getCoreBinder(seq,contributionScore,applyDataName=
    None,saveCoreBinders=False)
```

Arguments

`seq`

The amino-acid sequences in a list. The sequences must all have the same length, i.e. use *int2aa* (section 2.1.4) on the output of *addEmptyPositions* (section 2.1.6).

`contributionScore`

The contribution score of each overlapping l-mer in all of the sequences to which the trained model has been applied, as given by the output of *feedForwardAndGetScore* (section 2.1.11).

`applyDataName`

The name of the sequences.

`saveCoreBinders`

Whether or not to save the core binders as an attribute.

Value
Returns `sCore`, a numpy array with the core binder of each sequence (as amino acids).

2.1.19 *save_object*

Description
Save with *pickle* the object class. It will be saved in the path *savePath* with the file name given as argument or by default *[allele]_ModelCNN.pkl*, where [allele] is the allele name.
In order to avoid loading problems if the object is loaded from another OS, the attribute *savePath* is deleted upon saving.
If the class contains a list of trained Keras neural networks, it will be deleted

as these nets are saved separately with the saving option of *trainCNN* (section 2.1.8).

Usage

```
CNNPepPred.save_object(name=None)
```

Arguments

`name`

Name of the file.

2.1.20 *load_object*

Description
Load another object class. This is meant to load previously trained models. As the attribute *savePath* is deleted upon saving (section 2.1.19), this function will reset it to be the parent directory of the argument *filename*.

Usage

```
CNNPepPred.load_object(filename)
```

Arguments

`filename`

Complete pathway to the object to load.

Value
Returns `obj`, the loaded object.

2.1.21 *feedForwardVisualization*

Description
Visualization of the feed-forward pass of the trained model on the set of sequences *s*. It will create a folder in the path *savePath* called *feed_forward_visualization* that will contain two folders: *nets* and *sequences*. The folder *nets* will contain a folder for each net of the trained model with each of its corresponding convolutional layer's filters and dense layer's weights represented as images. The folder *sequences* will contain one folder for each of the input argument

sequences with an image of their encoding and a folder for each net containing the convolutional layer's output and the maxpooling layer's output represented as images.
For each input sequence, many images will be saved; it is therefore recommended to only run this function on a small pre-selected set of sequences.
For more information on the visualization of the feed-forward pass, see Appendix A.3.

Usage

```
CNNPepPred.feedForwardVisualization(s,fontSize=4,dpi=300)
```

Arguments

`s`

The amino acid sequences in a list.

`fontSize`

The font size of the x and y tick labels. Default is 4.

`dpi`

The dpi of the images. Default is 300.

Value
Returns `yhat`, a numpy array with the predicted outcome of each sequence.

2.1.22 *generateCVpartWithLeastLmerOverlap*

Description
Generate a cross-validation partition for the training data such that the number of shared l-mers between folds is reduced, where l is the length of the core binders as given in the parameter file (section 2.2).
For more information on the way the partition is generated, see Appendix B.2.

Usage

```
CNNPepPred.generateCVpartWithLeastLmerOverlap(kFold,saveCVPart=
    False)
```

Arguments

`kFold`

The number of folds, as an integer.

`saveCVPart`

Whether or not to save the cross-validation partition as an attribute of the class called *cvPart*. If true, the average number of shared l-mers between each of the `kFold` train/test partitions (within each positive and negative class) will also be saved as an attribute of the class called *averageLmersOverlappingCV*.

Value
Returns `cvPart`, a numpy array with the cross-validation partition and `averageLmersOverlappingCV`, the average number of shared l-mers between each of the `kFold` train/test partitions (within each positive and negative class).

2.2 The parameter file

When initializing the class, the parameters will be set from the file given with full path in the template or, by default, the file in the working directory called *parameters.txt*. This file consists of two columns (separated by a comma), one with the name of the parameter and one with the value of the parameter. Only the parameter values can be changed if needed. If a parameter value is left empty, the default value will be set (if left empty, check that the comma separating the columns is still there). The parameters are the following.

- *bindingThr*. Default: 0.5.
  The binding threshold for the predicted values.

- *similarityMat*. Default: blosum62.txt
  The similarity matrix to use for the sequence encoding. It must be symmetric and be of the same format, with the same amino-acid order, as the default file.

- l. Default: 9
  The length of the core binder.

- *maxNbSamples2apply*. Default: 50000
  The maximum number of sequences on which a trained model can be

applied in one batch. This is only for the application of the model through the function *feedForwardAndGetScore* (section 2.1.11). Increase if you have enough memory and decrease if you don't have enough memory.

- *nbPrev*. Default: 2
  The number of empty positions (corresponding to the absence of amino acids) to add at the beginning of a sequence.

- *nbAfter*. Default: 2
  The number of empty positions (corresponding to the absence of amino acids) to add at the end of a sequence.

- *F*. Default: 5/10/20/30
  The number of filters of the convolutional layer. Different number of filters can be given, separated by a slash "/". In that case the final model will be an -equally weighted- ensemble of models with different number of filters.

- *rep*. Default: 10
  The number of models to train with different initial weights per number of filters. For each number of filters given in the parameter *F*, *rep* number of models will be trained. The final model will be an equally weighted ensemble of *rep* times the number of different number of filters, i.e. $40 = 10 \cdot 4$ with the default parameters.

- *nMaxPool*. Default: see Appendix A.2.
  The pooling size of the Maxpooling layer will be *nMaxPool*×1. The default value is set by a formula given in the Appendix A.2 and will be such that the output layer has size $L_{freq} \times F$ where L_{freq} is the most frequent sequence length in the training data set and F is the number of filters.

- *initializeStd*. Default: 0.01
  The standard deviation of the initial weights (randomly generated from the normal distribution with zero mean). The same value will be used for the convolutional and the dense layers.

- *alpha*. Default: 0.005
  The learning rate of the stochastic gradient descent.

If transfer learning is used for training, two different values can be given (one for each optimization step of the training, see Appendix A.5). The two values must be separated by a slash "/", for example "0.005/0.001". If only one value is given, it will be used for both optimization steps.

- *gamma*. Default: 0.9
  The momentum of the stochastic gradient descent.
  If transfer learning is used for training, two different values can be given (one for each optimization step of the training, see Appendix A.5). The two values must be separated by a slash "/", for example "0.9/0.5". If only one value is given, it will be used for both optimization steps.

- *l2_fact*. Default: 0.0001
  The L2 regularization factor. The same value will be used for the convolutional and the dense layers.

- *maxEpochs*. Default: 30
  The number of epochs.
  If transfer learning is used for training, two different values can be given (one for each optimization step of the training, see Appendix A.5). The two values must be separated by a slash "/", for example "20/10". If only one value is given, it will be used for both optimization steps. If 0 is given for the second value (i.e. "20/0"), the second optimization will be skipped.

- *miniBatchSize*. Default: 128
  The size of the mini batch for the stochastic gradient descent.
  If transfer learning is used for training, two different values can be given (one for each optimization step of the training, see Appendix A.5). The two values must be separated by a slash "/", for example "128/56". If only one value is given, it will be used for both optimization steps.

- *useBias*. Default: 1
  Whether or not to use bias. The same value will be used for the convolutional and the dense layers.

- *activationFctDenseLayer*. Default: linear
  The activation function of the last layer (the *Dense* layer). Possible values are to choose among *keras*'s activation functions.

- *lossFct*. Default: mean_squared_error
  The loss function. Be aware that if changed, some parameter tuning might be needed. For example if for a classification problem you would rather use the *binary_crossentropy* loss function, you should change the activation function of the *Dense* layer to be the *sigmoid* function. Possible values are to choose among *keras*'s loss functions.

- *nbRandSeq*. Default: 200000
  The number of random sequences to be generated in the function *generateRandomSeq* (section 2.1.12).

- *nbBest*. Default: 2000
  The number of best scoring sequences to select for the generation of the logo plot with *plotLogoSeq* (section 2.1.13)

- *lengthRandSeq*. Default: 15
  The length of the random sequences generated in the function *generateRandomSeq* (section 2.1.12).

2.3 The template file

Fill the template file given in the main folder *CNN-PepPred* according to the desired analysis. This template consists of two columns (separated by a comma), one with the name of the template's inputs and one with their values. Only the input values can be changed if needed. If an input value is left empty, the default value will be set (if left empty, check that the comma separating the columns is still there). The inputs are the following.

- *allele*.
  The name of the allele. This name can be thought of as a job name for the run. If the training option is not selected and no trained model is given as input, then *allele* corresponds to the name of a pre-trained model (section 2.5).

- *savePath*. Default: os.getcwd()
  The pathway where to save the results.

- *doTraining*. Default: 0
  Whether or not to do the training.

- *trainingDataPath*. Default: None
  The file with the training data. It must be a *.txt* file, with at least two columns (with headers) separated by a comma. The first column contains the sequences and the second the outcome. For regression, the outcome must be already normalized. A third column containing a cross-validation partition can be added. If the cross-validation option is selected and no partition is given here, it will be generated following the function *generateCVpartWithLeastLmerOverlap* (section 2.1.22).
  If training data are given and a previously trained model is given in the template as *trainedModelsFile*, transfer learning will be used for training (see Appendix A.5).

- *doLogoSeq*. Default: 0
  Whether or not to plot (logo plot) the core binding pattern of the trained model.

- *doCV*. Default: 0
  Whether or not to do the cross-validation.

- *kFold*. Default: 5
  The number of folds for the cross-validation. If a partition is given in the training data file, this input will be ignored and the *kFold* value will be the number of partitions.

- *doApplyData*. Default: 0
  Whether or not to apply the trained model to new sequences.

- *trainedModelsFile*. Default: None
  Either the file containing the trained model (a *.pkl* file) or the pathway of the folder containing the *parameters* file and the *nets* folder with the trained nets (as saved with the function *trainCNN*, see section 2.1.8). If the input is a *.pkl* file, the parent folder must contain the *nets* folder. This option is only valid if no training is selected.
  If the apply or the logoseq option are selected with no training and *trainedModelsFile* is left empty, then a pre-trained model will be selected based on the *allele*. For available alleles, see section 2.5.
  If a previously trained model is given and training data are given in the template as *trainingDataPath*, transfer learning will be used for training (see Appendix A.5).

If a previously trained model is given as input, only the values of *nbRandSeq*, *nbBest*, *lengthRandSeq* and *maxNbSamples2apply* of the parameter file *parametersFile* given in the template will be considered. If transfer learning is used for training, the values of *alpha*, *gamma*, *maxEpochs*, *miniBatchSize*, *activationFctDenseLayer*, *lossFct*, *initializeStd* will also be considered. The remaining parameter's values will be taken from the parameter file of the previously trained model.

- *applyDataPath*. Default: None
  The file containing the data on which the trained model will be applied. It must be a FASTA file.

- *epitopesLength*. Default: 15
  The length of the epitopes on which the trained model will be applied. Each new sequence will be cut into all overlapping *epitopesLength*-mers and a prediction will be made for each of them.

- *parametersFile*. Default: parameters.txt (in the working directory)
  The full path for the file containing the parameters of the model. Parts of this file are ignored if a trained model is given as input in *trainedModelsFile*.

- *saveClassObject*. Default: 0
  Whether or not to save the class generated following the template in *savePath*. If the class contains a list of trained Keras neural networks, it will be deleted as these nets are saved separately with the saving option of *trainCNN* (section 2.1.8).

2.4 The script *model_from_template.py*

The argument of the script *model_from_template.py* is the template file. By default this file is called *template.txt* and is located in the working directory, the name and pathway can be modified but need to be given with full path as a system argument.
The script will first read the system argument to obtain the name of the template and call the main function with this template as an argument.

```
tmplName = sys.argv
if len(tmplName)==1:
```

```
   tmplName = 'template.txt'
else:
   tmplName = tmplName[1]
main(tmplName)
```

The *main* function will run the desired analysis following the template. First, the start time is recorded and the template is read,

```
time_start = time.perf_counter()
file = Path(tmplName)
allele,savePath,doTraining,trainingData,trainingOutcome,
   doLogoSeq,doCV,cvPart,kFold,doApplyData,trainedModelsFile,
   applyData,applyDataName,epitopesLength,parametersFile,
   saveClassObject = readTemplate(file)
```

then, the class *CNNPepPred* is initialized

```
modelCNN = CNNPepPred(allele,savePath,doTraining,trainingData,
   trainingOutcome,doLogoSeq,doCV,cvPart,kFold,doApplyData,
   trainedModelsFile,applyData,applyDataName,epitopesLength,
   parametersFile)
```

and the desired analysis will be performed following the template. If the training option is selected, the images *IM* encoding the sequences and training outcome *out* are first retrieved.

```
sInt = modelCNN.aa2int(modelCNN.trainingData)
modelCNN.seqLength(sInt,saveOutput=True)
sInt = modelCNN.addEmptyPositions(sInt)
IM = modelCNN.getImages(sInt)
out = modelCNN.trainingOutcome
```

Cross-validation with the training data is performed as follows:

```
modelCNN.crossValCNN(IM,out)
modelCNN.getCVresults()
```

The final model, to be saved in the object *modelCNN*, will be trained with all of the training data.

```
modelCNN.trainCNN(IM,out,saveModel=True)
```

To obtain the logoplot with the binding core, the script generates random sequences,

```
sR = modelCNN.generateRandomSeq()
```

applies the model to obtain the predicted outcomes and contribution scores of the random sequences' overlapping *modelCNN.l*-mers

```
contributionScore,yhatR = modelCNN.feedForwardAndGetScore(sR)
```

and finally generates the logoplot.

```
modelCNN.plotLogoSeq(contributionScore,yhatR)
```

The sequences on which the trained model must be applied are first cut into all the overlapping *epitopesLength*-mers.

```
sIntApply,sApplyName = modelCNN.seq2Lmer(modelCNN.aa2int(
   modelCNN.applyData),L=None,nameSeq=modelCNN.applyDataName,
   saveLmer = True)[0:2]
```

Then the amino-acid sequences are prepared in the required format for the application of the trained model.

```
sIntApply = modelCNN.addEmptyPositions(sIntApply)
```

The trained model is then applied to obtain the predicted outcomes and the contribution scores, which are used to find the binding cores, and the results are printed in the saving pathway.

```
modelCNN.feedForwardAndGetScore(sIntApply,saveOutcome = True)
modelCNN.getCoreBinder(modelCNN.int2aa(sIntApply),modelCNN.
   contributionScore,sApplyName,saveCoreBinders = True)
modelCNN.printApplyOutcome()
```

Finally the computation time is saved in the object and the object is saved in the saving pathway if selected in the template.

```
time_elapsed = (time.perf_counter() - time_start)
modelCNN.computationTime(time_elapsed)
if saveClassObject:
   modelCNN.save_object()
```

2.5 The pre-trained models

The user can use models available for some alleles which were trained with IEDB data (Appendix B.1). The models are in a folder called *trainedIEDB-models* of the main directory *CNN-PepPred*. In this case, the template must

contain the name of the allele and the data to apply the model to; no training must be selected and the trained model file (*trainedModelsFile*) must be left empty. An example template called *template_pretrained_model_example.txt* in the main directory has been pre-filled for the allele *HLA_DRB1_01_01*. The location where to save the results and the fasta file on which to apply the pre-trained model must be filled (*[your_path_to_save_the_results]* and *[fasta_file_for_prediction]* in the example template).
Available alleles are:

HLA_DPA1_01_03_DPB1_02_01,	HLA_DPA1_01_03_DPB1_03_01,	HLA_DPA1_01_03_DPB1_04_01,
HLA_DPA1_01_03_DPB1_04_02,	HLA_DPA1_01_03_DPB1_06_01,	HLA_DPA1_01_03_DPB1_104_01,
HLA_DPA1_02_01_DPB1_01_01,	HLA_DPA1_02_01_DPB1_09_01,	HLA_DPA1_02_01_DPB1_10_01,
HLA_DPA1_02_01_DPB1_14_01,	HLA_DPA1_02_01_DPB1_17_01,	HLA_DPA1_02_01__DPB1_13_01,
HLA_DPA1_02_02_DPB1_05_01,	HLA_DQA1_01_01_DQB1_05_01,	HLA_DQA1_01_02_DQB1_05_01,
HLA_DQA1_01_02_DQB1_06_02,	HLA_DQA1_02_01_DQB1_02_02,	HLA_DQA1_02_01_DQB1_03_01,
HLA_DQA1_03_01_DQB1_03_02,	HLA_DQA1_03_02_DQB1_04_01,	HLA_DQA1_05_01_DQB1_02_01,
HLA_DQA1_05_01_DQB1_03_01,	HLA_DQA1_05_05_DQB1_03_01,	HLA_DRB1_01_01,
HLA_DRB1_03_01,	HLA_DRB1_04_01,	HLA_DRB1_04_02,
HLA_DRB1_04_04,	HLA_DRB1_04_05,	HLA_DRB1_07_01,
HLA_DRB1_08_01,	HLA_DRB1_08_02,	HLA_DRB1_09_01,
HLA_DRB1_10_01,	HLA_DRB1_11_01,	HLA_DRB1_11_03,
HLA_DRB1_12_01,	HLA_DRB1_13_01,	HLA_DRB1_13_02,
HLA_DRB1_13_03,	HLA_DRB1_14_01,	HLA_DRB1_14_54,
HLA_DRB1_15_01,	HLA_DRB1_16_01,	HLA_DRB3_01_01,
HLA_DRB3_02_02,	HLA_DRB3_03_01,	HLA_DRB4_01_01,
HLA_DRB4_01_03,	HLA_DRB5_01_01,	HLA_DRB5_02_02.

2.6 Random generation of non-binders

The majority of experimental results only report binding peptides, so that most sets are too imbalanced to properly train a model. Therefore, we provide a separate script for the generation of randomly selected peptides that act as non-binders.
The script will simply select peptides at random from a user given folder containing fasta files, respecting the length distribution of the binders in the training set. These files should contain enough natural random sequences so that there shouldn't be any patterns that would relate them to one another (e.g. a full proteome).

The script is in the main folder *CNN-PepPred*, it is called *generateRandomNonBinders.py* and contain a unique function with the same name. Therefore, to import it, use

```
from generateRandomNonBinders import generateRandomNonBinders
```

The function is

```
generateRandomNonBinders(fastaSeqLoc,seqL=None,seq=None,prop=1,
    N=None,maxFiles=None)
```

with arguments:

`fastaSeqLoc`

The location of the folder containing the fasta files to select from.

`seqL`

A numpy array with the lengths of the binding peptides in the training set.

`seq`

A list of amino-acid sequences corresponding to the binding peptides in the training set. If `seqL` is not given, it will be computed from this list. If `seqL` is given, this argument is ignored.

`prop`

The proportion of peptides to select. The number of selected peptides will be around `prop`·`N` where `N` is either the number of binding peptides or the argument `N`.
`prop` is 1 by default.

`N`

The number of peptides to select. The final number will be `prop`·`N`. Note that due to the nature of the algorithm, it is possible that the number of peptides in the output differs slightly from this number.
If no sequences or length of sequences is given, `N` will be 2000 by default.

`maxFiles`

The maximum number of files to read in the given folder `fastaSeqLoc`.
We recommend dividing the sequences to select from into many files in `fastaSeqLoc` and using the parameter `maxFiles` instead of having one big file. In this way, the computational time will be lower since the algorithm

will only read few smaller files rather than a big one and there won't be any memory issues.

The function will return `seqNeg`, a list of amino-acid sequences respecting the number of sequences and their length distribution according to the given input arguments.
The function can be called as follows, with `myfolderwithsequences` being the pathway to the folder containing the sequences to select from.

```
seqNeg = generateRandomNonBinders(myfolderwithsequencesaSeqLoc,
   seqL=bindersLength,prop=1.3,maxFiles=3)
```

In this case the output `seqNeg` will have around 1.3 times the number of elements in `bindersLength`, with lengths ditributed like in `bindersLength` and selected from 3 randomly selected fasta files in the folder `myfolderwithsequences` . On the other hand

```
seqNeg = generateRandomNonBinders(myfolderwithsequencesaSeqLoc,
   seq=bindersSeq,N=2500,maxFiles=1)
```

will return around 2500 peptides respecting the length distribution of the sequences in `bindersSeq` and randomly selected from 1 file in the folder `myfolderwithsequences`.

3 Examples

Three different templates were prepared as examples in the main folder *ModelCNN*. To use them, you will need to change the pathways in the templates adapting them to the operating system of your computer and replace *[your_working_path]* by the pathway of the folder *ModelCNN*.
To run the template files, set your working directory to *ModelCNN* and type in your console

```
import sys
model_from_template = open("model_from_template.py").read()
sys.argv = ['model_from_template.py','full_path_to_any_template
    .txt']
exec(model_from_template)
```

Or, alternatively,

```
import model_from_template
modelCNN = model_from_template.main('full_path_to_any_template.
    txt')
```

3.1 Template 1: Train+CV+logoPlot+Apply

The first template, *template1_Train_CV_logoPlot_Apply.txt*, will perform cross-validation and train a model using the example training data set of allele *HLA_DRB1_08_01* in the folder *Example*. It will also generate the logo plot representing the binding characteristics of the trained model and apply it to new sequences *uniprot-proteome_UP000000605_100.fasta* in the same folder. The results will be saved in the folder *Template1_results* of the *Example* folder.
Here are the cross-validation results obtained after runing this template (note that there might be small differences between runs):

```
Allele,#Peptide,#Binder,PC,AUC,RMSE,MCC,ACC,BACC,F1
HLA_DRB1_08_01
    ,1118,559,0.783,0.962,0.312,0.834,0.917,0.917,0.917
```

The different scores are: PC (Pearson correlation), AUC (area under the curve), RMSE (root mean square error), MCC (Matthews correlation coefficient), ACC (accuracy), BACC (balanced accuracy), F1 (F1-score).
Figure 1 contains the logo plot of the trained model:

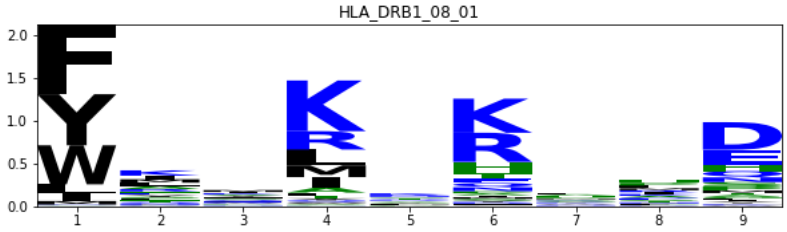


Figure 1: Logo plot of the first template.

Here is a list of some of the highest predicted binders:

```
Peptide_Source,Start,End,Peptide,Binding_Core,Predicted_Outcome
spQ63PT2SAHH_BURPS,168,182,EVALFKSIERHLEID,FKSIERHLE,1.454
spQ63Q03RPOB_BURPS,1069,1083,VKVYLAVKRRLQPGD,YLAVKRRLQ,1.392
spQ63UT2SYH_BURPS,346,360,REQAFIVAERLRDTG,FIVAERLRD,1.375
spQ63PT2SAHH_BURPS,103,117,GTPVFAFKGESLDEY,FAFKGESLD,1.371
spQ63NC4ACSA_BURPS,572,586,VVAFVVLKRSRPEGE,FVVLKRSRP,1.327
spQ63Y06SYR_BURPS,440,454,AVRFFLISRKADTEF,FFLISRKAD,1.303
spQ63WM0RS20_BURPS,28,42,FRTAIKAVRKAIDAG,IKAVRKAID,1.289
spQ63WM0RS20_BURPS,47,61,AAELFKAATKTIDTI,FKAATKTID,1.278
spQ63TM2SYT_BURPS,575,589,EKISYKIREHTLEKV,YKIREHTLE,1.236
spQ63UY6RS6_BURPS,85,99,LRHLIVKMKKAETGP,LIVKMKKAE,1.232
```

The first column is the name of the sequence, as written in the FASTA file. The second and third columns are respectively the start and end position of the peptide in the sequence. The fourth column is the peptide and the fifth column its binding core. The sixth colum is the model's predicted outcome.

3.2 Template 2: Train

The second template, *template2_Train.txt*, will train a model using the example training data set of allele *HLA_DRB1_08_01* in the folder *Example*. The results will be saved in the folder *Template2_results* of the *Example* folder.

3.3 Template 3: Apply with template 2 trained model

The third template, *template3_Apply.txt*, applies the pre-trained model of *HLA_DRB1_08_01* to new sequences

uniprot-proteome_UP000000605_100seq.fasta in the *Example* folder. The results will be saved in the folder *Template3_results* of the *Example* folder. Here is a list of some of the highest predicted binders:

```
Peptide_Source,Start,End,Peptide,Binding_Core,Predicted_Outcome
spQ63PT2SAHH_BURPS,168,182,EVALFKSIERHLEID,FKSIERHLE,1.449
spQ63UT2SYH_BURPS,346,360,REQAFIVAERLRDTG,FIVAERLRD,1.391
spQ63Q03RPOB_BURPS,1069,1083,VKVYLAVKRRLQPGD,LAVKRRLQP,1.369
spQ63PT2SAHH_BURPS,103,117,GTPVFAFKGESLDEY,FAFKGESLD,1.351
spQ63Y06SYR_BURPS,440,454,AVRFFLISRKADTEF,FFLISRKAD,1.327
spQ63WM0RS20_BURPS,29,43,RTAIKAVRKAIDAGD,IKAVRKAID,1.309
spQ63NC4ACSA_BURPS,572,586,VVAFVVLKRSRPEGE,FVVLKRSRP,1.305
spQ63T53ALLC1_BURPS,34,48,DDFFAPKERMLNPEP,FAPKERMLN,1.301
spQ63WM0RS20_BURPS,47,61,AAELFKAATKTIDTI,FKAATKTID,1.272
spQ63TM2SYT_BURPS,575,589,EKISYKIREHTLEKV,YKIREHTLE,1.269
```

Appendix A: A convolutional neural network architecture for the prediction of peptide's binding

A.1 Peptide's encoding

The peptides are encoded using the *blosum62* similarity matrix [4].

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

The symbol "-" stands for the absence of amino acids. A similar type of encoding is used in the models of the netMHCII family [5].
With the template, you can set your own similarity matrix keeping the above format and amino acid order.
A peptide will then be encoded as an "image" for the input of the convolutional neural network. This image can be though of as a table where the rows are the residues of the peptide and the columns are the 20 amino acids

+ the absence of amino acid. This table is then filled using the corresponding similarity value. To account for the difference in peptides' lengths, the absence-of-amino-acid character "-" will be added at the end of each peptide until its length matches the maximal length in the training data set. Moreover, a fixed number of character "-" will be added at the beginning and end of the peptide; this step can be thought of as a sequence equivalent to an image zero-padding. The exact number of additional characters "-" at the beginning and end of the sequence is determined by the parameters *nbPrev* and *nbAfter*, repsectively; they are both set to 2 by default. Therefore, the number of rows, i.e. the length of the input peptide, will be the length of the maximal peptide in the training data set + *nbPrev* + *nbAfter*.
For example, the peptide *MSAIESVLHERRQFA*, in a model where the maximal length is 20, will be encoded as:

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

```
-,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,1
```

A.2 The model's architecture

The neural networks are implemented as *Keras* [3] sequential models with the following architecture:

1. Convolutional layer with *ReLu* activation.
2. Maxpooling layer.
3. Dense (or fully connected) layer with parameter-defined activation function.

The initial weights of both the first and third layers are randomly generated from a normal distribution with zero mean and a standard deviation defined by the parameter *initializeStd* (0.01 by default).
The filters of the convolutional layers are of size $l \times 21$ where l (the length of the binding core) is defined in the parameter file (9 by default) and 21 are the 20 amino acids plus the absence of amino acid. The strides are of size 1×21, so that, in practice, the filters will only convolute along the first dimension with stride 1. The model therefore optimizes the search for l-mer core binders contained within a peptide.
The number of filters is defined in the parameter file. Multiple different numbers can be chosen and *rep* neural networks will be trained for each number of filters, where *rep* is the number of repetitions so that the final model is trained from different initial configurations. By default, the final model will be an equally weighted ensemble of 40 neural networks: 10 of them with 5 filters, 10 with 10 filters, 10 with 20 filters and 10 with 30 filters.
The pooling size of the maxpooling layer is of size $m \times 1$ with stride 1×1. The parameter m can be set as *nMaxPool* in the parameter file (section 2.2). By default m is defined as follows:

$$m := \max(\{6, L_{max} - l - L_{freq} + nbPrev + nbAfter + 2\})$$

where L_{max} is the maximal peptide length in the training data set, L_{freq} the most frequent one and *nbPrev* and *nbAfter* are the number of characters "-" added at the beginning and end of each sequence (see A.1). The formula for m is defined to make sure that it will neither be too small (the minimal

value is 6) nor too big compared to L_{max} which changes from data set to data set; it will control the size of the maxpooling layer's output to be equal to $L_{freq} \times F$, where F is the number of filters. Indeed, the height of the input image is $h := L_{max} + nbPrev + nbAfter$; therefore, the size of the convolutional layer's output is $(h - l + 1) \times F$ and the size of the maxpooling layer's output is $(h - l - m + 2) \times F = L_{freq} \times F$. However, m has a minimum value of 6 to avoid having a pool size too small, so the first dimension of the maxpooling layer's output might be smaller.
The weight optimization is done with a mini batch stochastic gradient descent with parameters defined in the parameters file (section 2.2).

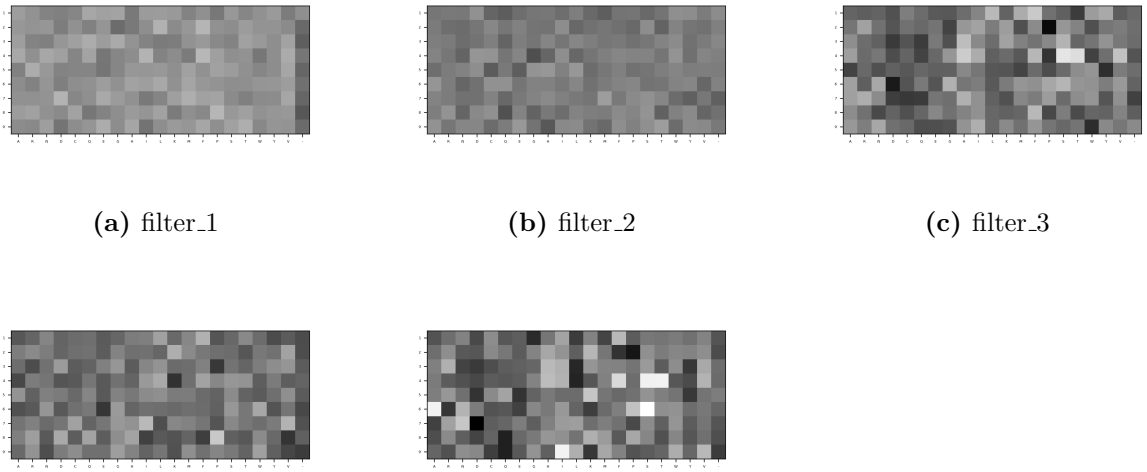
A.3 Visualization of the feed-forward pass

Calling the function *feedForwardVisualization* (section 2.1.21) will generate images to visualize the feed-forward pass of each net in the trained model on the sequences given as input of the function. The results will be saved in a folder called *feed_forward_visualization* in the saving pathway *savePath* of the class.
You can call this function through a class with a trained model. It can be applied to a previously trained and saved model in the following way:

```
from model_initializer import CNNPepPred
path_to_trained_model = ...
s = ['KPTHFTVLTKGAGK', 'SEIQYKILTQKEDD', 'TAVFLAAGVGMRL']
myModel = CNNPepPred(trainedModelsFile=path_to_trained_model,
   doApplyData=True,applyData=s)
yhat = myModel.feedForwardVisualization(myModel.applyData)
```

where *path_to_trained_model* is either a *.pkl* saved class object or the path of the saved model (like the input *trainedModelsFile* in the template, see section 2.3).

In Figure 2 we present an example of this visualization for the 5 filters of the convolutional layer of one net.



(a) filter_1 **(b)** filter_2 **(c)** filter_3

(d) filter_4 **(e)** filter_5

Figure 2: The 5 filters of the convolutional layer.

These filters are of size 9×21, where the columns are the amino acids *ARNDCQEGHILKMFPSTWYV-* and 9 is the length of the l-mers that the filters will highlight. Pixels with higher values are in white.
The weights in the filters could be thought of as PSSM matrices highlighting particular amino acids at given positions within a nonamer. For example, the filter (e) in Figure 2 will activate nonamers containing the amino acids S and T in position 4, A and S in position 6 and I in position 9 (and to some extent F in position 1 and N in position 7).
The encoded image of the peptide sequence *VLVKEIRSLGIDIDL* is printed below in Figure 3,

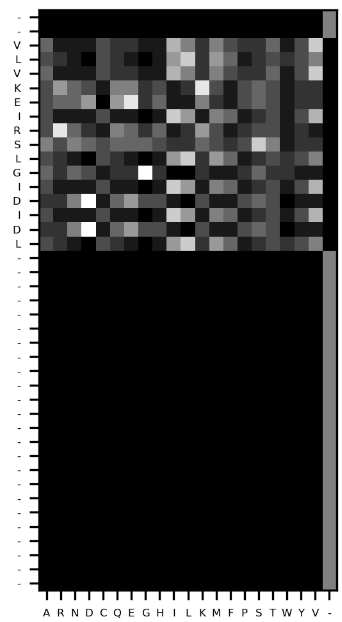


Figure 3: The encoded input image

with *nbPrev* and *nbAfter* equal to 2 and the maximal length in the training data set being 37.
After the convolution of the filters on the peptide's encoding image, the output is printed in Figure 4,

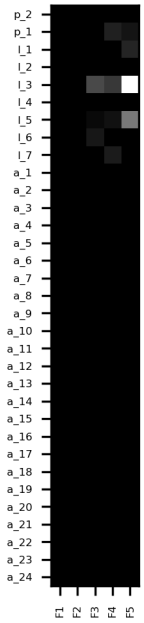
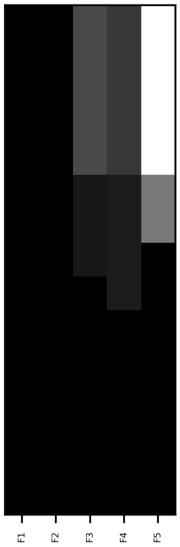


Figure 4: The output of the convolutional layer.

where each column represents the output after the convolution of each filter $F1$, $F2$, $F3$, $F4$ and $F5$. The rows corresponds to the application of each filter on an overlapping nonamer of the sequence - -*VLVKEIRSLGIDIDL*- -. The overlapping nonamers of the original peptide *VLVKEIRSLGIDID* are labelled l_x where x is the start position of the nonamer in this sequence. The overlapping nonamers containing the characters '-' that were added before the first residue of the original peptide are labelled p_x, where x is an integer that decreases when approaching the first nonamer fully composed of the original peptide's residues, i.e. without special characters added. Similarly, the overlapping nonamers containing the added characters after the last residue of the peptide are labelled a_x, where x increases when moving away from the last nonamer fully composed of the original peptide's residues.
The output of the maxpooling layer is printed in Figure 5,



F1	F2	F3	F4	F5
p_2	p_2	l_3	l_3	l_3
p_1	p_1	l_3	l_3	l_3
l_1	l_1	l_3	l_3	l_3
l_2	l_2	l_3	l_3	l_3
l_3	l_3	l_3	l_3	l_3
l_4	l_4	l_6	l_7	l_5
l_5	l_5	l_6	l_7	l_5
l_6	l_6	l_6	l_7	l_6
l_7	l_7	l_7	l_7	l_7
a_1	a_1	a_1	a_1	a_1
a_2	a_2	a_2	a_2	a_2
a_3	a_3	a_3	a_3	a_3
a_4	a_4	a_4	a_4	a_4
a_5	a_5	a_5	a_5	a_5
a_6	a_6	a_6	a_6	a_6

Figure 5: Left: the output of the MaxPool layer. Right: the argument of each pixel in the output.

where the image on the left is the output of the maxpooling layer with each column corresponding to a filter and the table on the right corresponds to the argument of each pixel in the image. In other words, each cell of the table corresponds to a pixel in the output image and the content of this cell is the overlapping nonamer (labelled as described above) that was selected during the maxpooling layer. The table will be saved as an html table.
Finally, the dense layer with the below weights (Figure 6) is applied to this output to obtain the net's prediction.

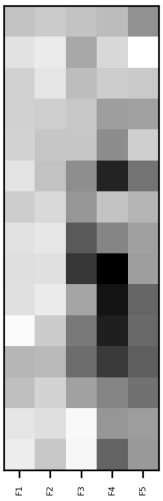


Figure 6: The weights of the dense layer.

Note that the biases of the net are not represented in this visualization. The feed-forward pass visualization will generate many images; it is therefore recommended to first select a small subset of peptides of interest and only then call the function with this subset.

A.4 The contribution score

Let l be the length of the core binder. We define here the *contribution score* associated to each of the overlapping l-mers of a peptide sequence. This score can be understood as the relative importance of an l-mer to the predicted outcome of the corresponding peptide.
For a fixed peptide, let s be any of its overlapping l-mers and we will give a brief description of how the model is applied with respect to s.
The first layer of the model is a convolutional layer with F filters and the corresponding output layer consists of one value for each overlapping l-mer and each filter, i.e. each filter is applied to each overlapping l-mer to obtain an output layer with size the number of overlapping l-mers times the number of filters. Let $x_i^{(s)}$ denote the output value after the application of the filter i to the l-mer s for $i = 1, ..., F$. The activation function of this layer is the

ReLu activation, i.e. $x_i^{(s)}$ will be mapped to 0 if it is negative and will remain unchanged otherwise. For ease of notation, let $x_i^{(s)}$ be the output value after the *ReLu* activation.
The second layer is a maxpooling layer, therefore only the maximal values will remain with possible repetitions, i.e. for filter i, the value of the application of the model so far to s will be $m_i \cdot x_i^{(s)}$ where m_i is a positive integer (including zero).
The final layer is a dense layer with a parameter-defined activation function σ. This layer will multiply all the values by the weights $d_i^{(s)}$ of the layer and sum them to obtain one remaining value. Keeping only the terms related to s, we define

$$w^{(s)} := \sum_i d_i^{(s)} \cdot m_i \cdot x_i^{(s)}$$

which corresponds to the application of the model restricted to s. In particular, the predicted outcome $\hat{y}$ will be

$$\hat{y} = \sigma\left(\sum_{s'} w^{(s')} + b\right)$$

where b is the bias of the dense layer.
Therefore, the relative contribution of s, with respect to the other l-mers, to the predicted outcome can be thought of as

$$\phi^{(s)} := \frac{w^{(s)}}{\sum_{s'} w^{(s')}}$$

where we define $\phi^{(s)}$ to be the contribution score of s. Note that this value can be smaller than 0 and bigger than 1.
The final model is an ensemble of N convolutional neural networks. Let $w_n^{(s)}$ be the above defined value for the net n and let b_n be its last layer's bias, then the predicted outcome is

$$\hat{y} = \frac{1}{N}\sum_n \sigma\left(\sum_{s'} w_n^{(s')} + b_n\right)$$

and we define the contribution score of s for an ensemble of nets to be

$$\phi^{(s)} := \frac{\sum_n w_n^{(s)}}{\sum_n \sum_{s'} w_n^{(s')}}.$$

Note that $\sum_{s'} \phi^{(s')} = 1$ and, if σ is the linear activation, then $\phi^{(s)} = \frac{\sum_n w_n^{(s)}}{N\hat{y} - \sum_n b_n}$. The predicted binding core, s_{core}, is then defined to be the overlapping l-mer of the peptipe with the highest contribution score, i.e.

$$s_{\text{core}} \in \text{argmax}_{s'}(\phi^{(s')}).$$

A.5 Transfer learning

Transfer learning uses previously trained models to solve new similar problems. A guide on transfer learning with Keras is available at: `https://keras.io/guides/transfer_learning/`.
In the context of MHC-class II peptide binding prediction, if a model trained for a given allele is available, transfer learning can be used to fit new data from another similar allele. In CNN-PepPred, transfer learning is implemented in two steps:

1. The convolution layer is extracted from the given previously trained model and is used as the corresponding layer for the new model. The weights of this layer are frozen, i.e. they will not be updated during the optimization. A first round of optimization is performed, where only the weights of the dense layer are updated.

2. The weights of the convolutional layer are unfrozen and a second round of optimization is performed where all weights are updated.

In the parameter file (section 2.2), two different values can be given for the optimization parameters (*alpha*, *gamma*, *maxEpochs*, *miniBatchSize*) each corresponding to the two different rounds of optimization. The second step can be skipped by setting the corresponding value of *maxEpochs* to 0.
Transfer learning can be used from the template (section 2.3) by inputting training data (in *trainingDataPath*) and a trained model (in *trainedModelsFile*).
Transfer learning can lead to overfitting and some considerations must be taken with the optimization parameters. It is recommended to reduce the number of epochs *maxEpochs* (for example setting it to 15) and/or reduce the learning rate *alpha* (for example to 0.001). The first step will correspond to the same model as the pre-trained model with a dense layer fitted to the new data. The second step will fine-tune this model to fit the new data. It is therefore possible that the final model differs from the pre-trained one if

the learning is too large and it can also lead to overfitting if the data from the two models are similar.
In Appendix B.4 and Appendix D, cross-validation and evaluation results using transfer learning with IEDB data (Appendix B.1) are presented.

Appendix B: IEDB data

B.1 Data preparation

We extracted the data from the IEDB web page `https://www.iedb.org/mhcdetails_v3.php` in the *Assays* tab with filters *Epitope Structure Type: Linear Epitopes*, *Host Organism: Homo sapiens (human)* and *assay-mhc_allele-mhc_Blass: II*. The outcome was taken from the column *qualitative_measure*, the sequences with value *Positive* and *Positive-High* were tagged as binders (1), the ones with value *Positive-Low* and *Positive-Intermediate* were ignored as they might be too weak binders and the rest of the sequences with value *Negative* were tagged as non-binders (0). For each allele's data set, if there were more non-binders than binders, a subset of non-binders was selected at random to balance the data set. If there were more binders than non-binders, the set was balanced using the script *generateRandomNonBinders.py* (section 2.6). This script generates a given number of non-binders selected from FASTA sequences in a given folder. The sequences used to randomly select non-binders were retrieved from `https://www.uniprot.org/uniparc/`. To improve the computational speed, we only downloaded some batches of sequences from UniParc, namely all the entries starting with *UPI00XX* where $XX = 00, 01, 02, ..., 10, 11$. There were then more than 70 millions sequences. In order to avoid repetitive sequences in an allele's data set, which can bias the training and testing of the model, for each of the unique overlapping 11-mers contained in one class (binding or non-binding) of the allele's data, only the shortest peptide containing the 11-mer was included. The length 11 was selected because it can remove most of the repetitive sequences without being as restrictive as the length 9 (i.e. the length of the binding core).
Moreover, the cross-validation partition was set to avoid testing with peptides containing too many nonamers also contained in the training data (see next subsection).
Only alleles containing at least 100 positive peptides were included.

B.2 Cross-validation result

We performed a k-fold cross-validation with k= 5 on the allele specific data retrieved as described in the previous subsection. The cross-validation partition was generated using a simple approach in order to reduce the number of l-mers present in both the training and testing data, where l is the length

of the core binder (l= 9 here).
First, a random cross-validation partition is generated. Then the l-mers shared between the training and testing splits of the random cross-validation partition (within each positive or negative class) are selected. Finally, the peptides containing each of the previously selected l-mers are re-assigned to the fold which occurs the most in the set of peptides sharing the same l-mer. In this way, the number of l-mers shared between folds is greatly reduced compared to a random assignment and all of the cross-validation partitions have a similar number of peptides. Note that this procedure doesn't guarantee that the folds won't share any l-mers. Such a procedure would likely be computationally expensive and could lead to very imbalanced partitions.
This procedure was implemented as *generateCVpartWithLeastLmerOverlap* (section 2.1.22) and if cross-validation is selected in the template and no partition is given with the training data, the model assigns a partition that is fixed before training using this procedure. Moreover, this function will also count the number of overlapping l-mers between each of the training and testing splits and return the average count per split; it will be saved as an attribute called *averageLmersOverlappingCV*.
The cross-validation results are reported in the table below with the following scores: AUC (area under the curve), MCC (Matthews correlation coefficient), ACC (accuracy), F1 (F1-score), where we used CNN-PepPred with default parameters, namely an ensemble of 10 nets per number of filters (5/10/20/30), totalling 40 nets.
We also include the results for the same cross-validation folds using the NNAlign-2.1 method (see Appendix C for more details on NNAlign). We trained NNAlign using 10 seeds with 5,10,20,30 hidden neurons and the option 'Impose amino acid preference at P1 during burn-in' set to true, the cross-validation partition was given as input and the rescaling of the outcome was set to "No rescale", since the outcome was binary. Note that while NNAlign was rather meant for regression on a quantitative outcome, our model was also set to optimize the mean squared error (this can be set in the parameters), so that it could have been used with the exact same parameters on a quantitative outcome, just like NNAlign.
The best scores are highlighted in bold. For all alleles except one (with respect to the MCC/ACC/F1 score), CNN-PepPred outperformed NNAlign-2.1.

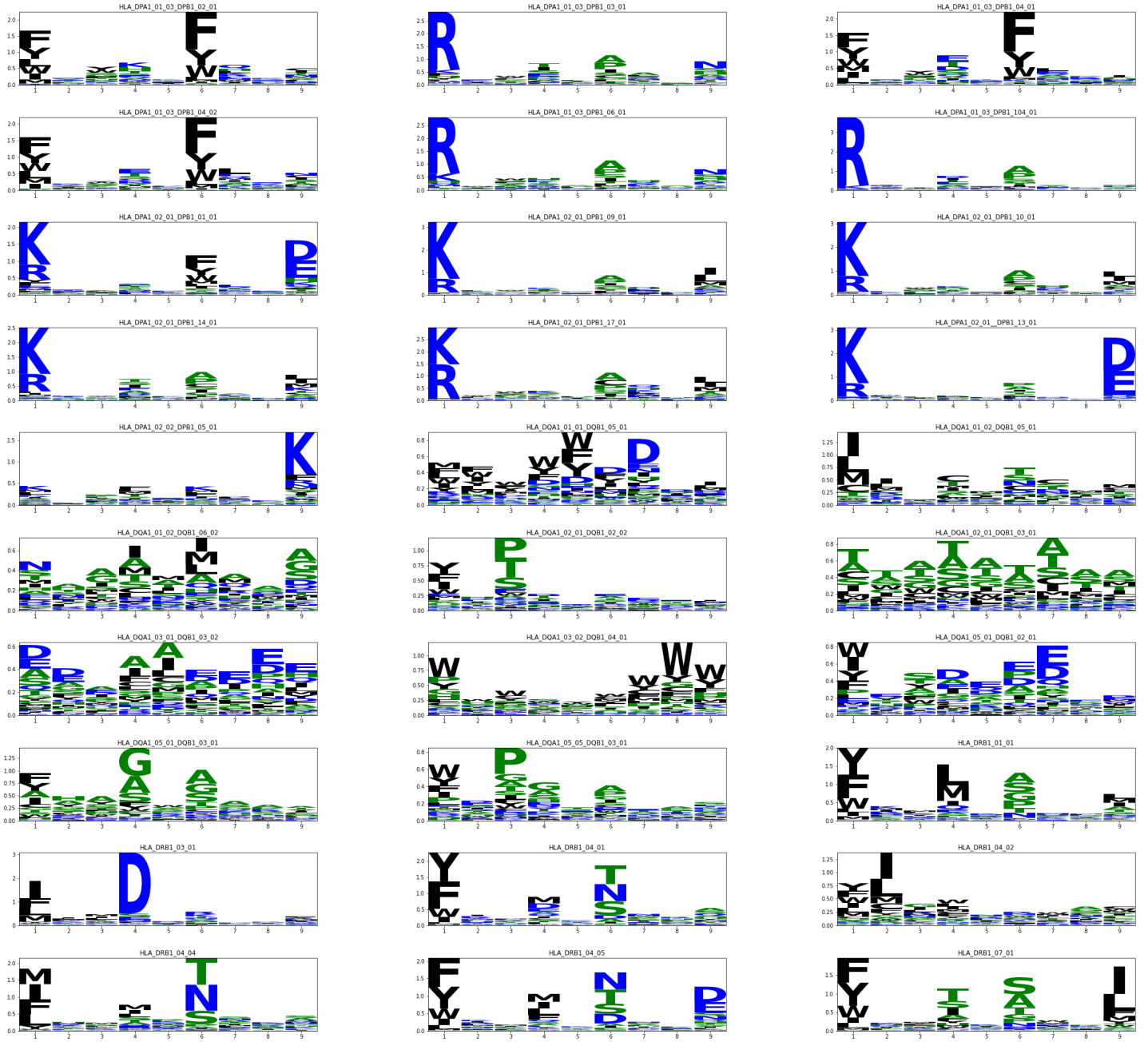
Allele	#Peptide	#Binder	CNN-PepPred				NNAlign-2.1			
			AUC	MCC	ACC	F1	AUC	MCC	ACC	F1
HLA_DPA1_01_03_DPB1_02_01	5177	2589	**0.951**	**0.763**	**0.88**	**0.874**	0.937	0.729	0.864	0.861
HLA_DPA1_01_03_DPB1_03_01	5025	2512	**0.959**	**0.803**	**0.901**	**0.898**	0.936	0.741	0.87	0.868
HLA_DPA1_01_03_DPB1_04_01	7984	3993	**0.946**	**0.747**	**0.872**	**0.865**	0.924	0.701	0.85	0.847
HLA_DPA1_01_03_DPB1_04_02	4643	2322	**0.971**	**0.839**	**0.919**	**0.917**	0.952	0.791	0.895	0.895
HLA_DPA1_01_03_DPB1_06_01	946	473	**0.965**	**0.784**	**0.891**	**0.887**	0.952	0.772	0.886	0.885
HLA_DPA1_01_03_DPB1_104_01	300	150	**0.99**	0.887	0.943	0.944	0.982	**0.9**	**0.95**	**0.95**
HLA_DPA1_02_01_DPB1_01_01	4292	2146	**0.969**	**0.829**	**0.914**	**0.913**	0.948	0.768	0.884	0.884
HLA_DPA1_02_01_DPB1_09_01	2692	1346	**0.972**	**0.835**	**0.917**	**0.916**	0.944	0.767	0.883	0.884
HLA_DPA1_02_01_DPB1_10_01	3628	1814	**0.97**	**0.822**	**0.911**	**0.909**	0.933	0.722	0.861	0.859
HLA_DPA1_02_01_DPB1_14_01	6035	3018	**0.966**	**0.808**	**0.904**	**0.902**	0.924	0.712	0.856	0.855
HLA_DPA1_02_01_DPB1_17_01	2170	1085	**0.974**	**0.839**	**0.919**	**0.918**	0.941	0.738	0.869	0.87
HLA_DPA1_02_01__DPB1_13_01	1968	984	**0.975**	**0.845**	**0.922**	**0.919**	0.969	0.826	0.913	0.913
HLA_DPA1_02_02_DPB1_05_01	7889	3945	**0.96**	**0.799**	**0.899**	**0.898**	0.922	0.708	0.854	0.854
HLA_DQA1_01_01_DQB1_05_01	208	104	**0.94**	0.741	0.87	0.867	**0.94**	**0.77**	**0.885**	**0.887**
HLA_DQA1_01_02_DQB1_05_01	410	206	0.764	0.362	0.68	0.668	**0.774**	**0.42**	**0.71**	**0.705**
HLA_DQA1_01_02_DQB1_06_02	1498	749	**0.915**	**0.672**	**0.835**	**0.829**	0.88	0.624	0.812	0.809
HLA_DQA1_02_01_DQB1_02_02	5772	2886	**0.901**	**0.653**	**0.826**	**0.82**	0.862	0.572	0.786	0.783
HLA_DQA1_02_01_DQB1_03_01	256	128	0.884	0.603	0.801	0.794	**0.904**	**0.664**	**0.832**	**0.83**
HLA_DQA1_03_01_DQB1_03_02	350	175	0.783	0.402	0.7	0.685	**0.795**	**0.475**	**0.737**	**0.729**
HLA_DQA1_03_02_DQB1_04_01	206	103	0.792	0.488	0.743	0.728	**0.815**	**0.564**	**0.782**	**0.789**
HLA_DQA1_05_01_DQB1_02_01	4051	2025	**0.872**	**0.574**	**0.786**	**0.776**	0.829	0.512	0.755	0.746
HLA_DQA1_05_01_DQB1_03_01	617	307	**0.909**	**0.668**	**0.833**	**0.825**	0.904	0.658	0.828	0.821
HLA_DQA1_05_05_DQB1_03_01	5882	2941	**0.889**	**0.63**	**0.815**	**0.811**	0.853	0.549	0.774	0.769
HLA_DRB1_01_01	12412	6208	**0.824**	**0.492**	**0.744**	**0.73**	0.795	0.445	0.722	0.711
HLA_DRB1_03_01	2178	1089	**0.866**	**0.553**	**0.775**	**0.763**	0.85	0.54	0.769	0.76
HLA_DRB1_04_01	5110	2557	**0.846**	**0.544**	**0.77**	**0.755**	0.834	0.533	0.765	0.752
HLA_DRB1_04_02	256	128	**0.764**	**0.469**	**0.734**	**0.73**	0.744	0.423	0.711	0.699
HLA_DRB1_04_04	3076	1538	**0.801**	**0.447**	**0.723**	**0.716**	0.754	0.376	0.687	0.675
HLA_DRB1_04_05	3972	1986	**0.913**	**0.676**	**0.837**	**0.83**	0.887	0.631	0.814	0.807
HLA_DRB1_07_01	4466	2233	**0.916**	**0.684**	**0.841**	**0.835**	0.907	0.675	0.837	0.834
HLA_DRB1_08_01	1118	559	**0.96**	**0.827**	**0.913**	**0.911**	0.957	0.806	0.903	0.904
HLA_DRB1_08_02	838	419	0.829	0.49	0.745	0.737	**0.839**	**0.523**	**0.761**	**0.758**
HLA_DRB1_09_01	1056	528	**0.906**	**0.672**	**0.836**	**0.835**	0.889	0.631	0.815	0.816
HLA_DRB1_10_01	2582	1291	**0.969**	**0.833**	**0.917**	**0.916**	0.963	0.825	0.912	0.913
HLA_DRB1_11_01	4180	2089	**0.917**	**0.665**	**0.832**	**0.826**	0.904	0.655	0.827	0.824
HLA_DRB1_11_03	422	211	**0.956**	**0.853**	**0.927**	**0.926**	0.95	0.801	0.9	0.901
HLA_DRB1_12_01	992	496	**0.966**	**0.809**	**0.904**	**0.903**	0.961	0.806	0.903	**0.903**
HLA_DRB1_13_01	1287	643	**0.935**	**0.74**	**0.869**	**0.865**	0.917	0.699	0.849	0.848
HLA_DRB1_13_02	1460	731	0.885	**0.607**	**0.803**	**0.802**	**0.886**	0.599	0.799	0.796
HLA_DRB1_13_03	1966	983	**0.986**	0.894	0.947	0.947	0.984	**0.896**	**0.948**	**0.948**
HLA_DRB1_14_01	681	340	**0.988**	**0.918**	**0.959**	**0.959**	0.971	0.88	0.94	0.94
HLA_DRB1_14_54	788	394	**0.998**	**0.959**	**0.98**	**0.98**	0.995	0.947	0.973	0.974
HLA_DRB1_15_01	4400	2201	**0.887**	**0.615**	**0.806**	0.796	0.876	0.607	0.803	**0.798**
HLA_DRB1_16_01	423	211	**0.959**	**0.822**	**0.91**	**0.907**	0.948	0.778	0.889	0.89
HLA_DRB3_01_01	1280	640	**0.951**	**0.819**	**0.905**	**0.899**	**0.951**	0.798	0.898	0.896
HLA_DRB3_02_02	1386	694	**0.979**	**0.865**	**0.932**	**0.931**	0.974	0.853	0.926	0.926
HLA_DRB3_03_01	210	105	**0.963**	**0.803**	**0.9**	**0.896**	0.956	0.784	0.89	0.895
HLA_DRB4_01_01	1316	658	**0.914**	**0.654**	**0.827**	**0.822**	0.897	0.628	0.814	0.812
HLA_DRB4_01_03	856	428	**0.971**	0.824	0.911	0.908	0.967	**0.832**	**0.916**	**0.916**
HLA_DRB5_01_01	3329	1664	**0.913**	0.676	0.837	0.833	0.911	**0.678**	**0.839**	**0.837**
HLA_DRB5_02_02	926	463	**0.989**	**0.92**	**0.96**	**0.96**	0.98	0.892	0.946	0.947
Average			**0.921**	**0.716**	**0.857**	**0.853**	0.907	0.691	0.845	0.843
Average weighted by #Binder			**0.919**	**0.702**	**0.85**	**0.845**	0.896	0.657	0.828	0.824

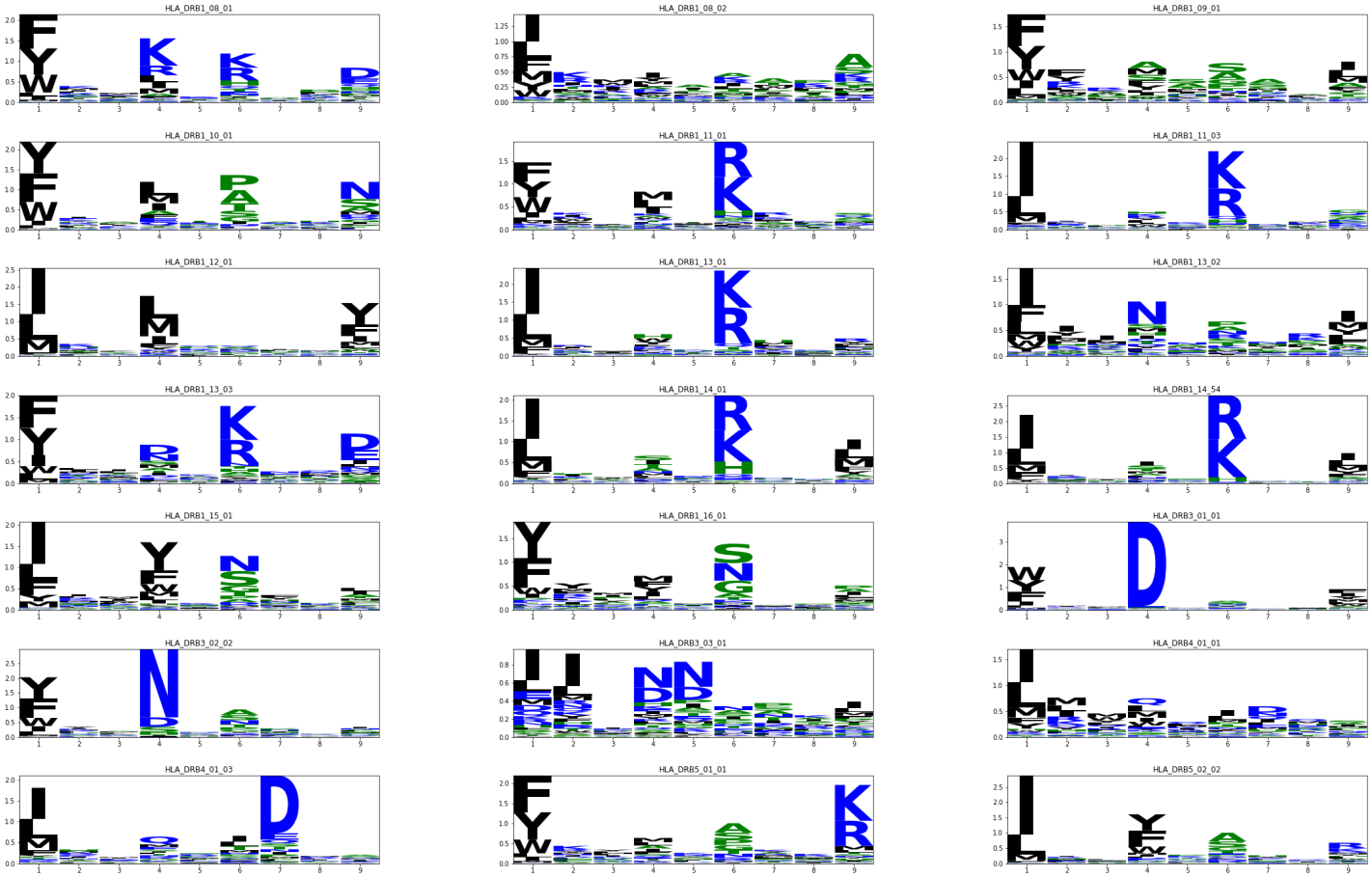
The table below shows the average number of shared nonamers between training/testing splits as given by the output *averageLmersOverlappingCV* of the function *generateCVpartWithLeastLmerOverlap* (section 2.1.22).
We also compared the computation times of CNN-PedPred (with GPU) and NNAlign for cross-validation. For CNN-PepPred, the total run time, including cross-validation, training on the full training data set and logo plot, is given in a separate column (called *total*) than the run time for cross-validation alone (called *cv*).
It can be observed that CNN-PepPred is generally faster for the alleles with a larger number of sequences and slower for the alleles with a smaller number of sequences. It also seems that with GPU, alleles that were computed towards the end (the order in the table corresponds to the chronological order of computation) used more time than those computed at the beginning. This is likely due to suboptimal implementation for consecutive runs, possibly in connection with the low-level GPU used (NVIDIA GeForce GTX 1080 under Windows OS). CPU runs were performed on the same computer, with an AMD Ryzen 7 1700 8-core, under Windows Susbsystem for Linux (WSL), since the NNAlign excecutable requires a Linux OS. We couldn't perform the GPU runs on WSL since the system does not support it. This benchmark is only indicative, since WSL is known to decrease performance by about 30% in average compared to native Linux and performance is anyway bound to the specific CPU and GPU used.

Allele	#Peptide	#Binder	Average number of shared nonamers between training/testing splits	CNN-PepPred		NNAlign-2.1
				time[s]: total	cv	time[s]
HLA_DPA1_01_03_DPB1_02_01	5177	2589	5.6	806	619	1610
HLA_DPA1_01_03_DPB1_03_01	5025	2512	0.8	811	624	1548
HLA_DPA1_01_03_DPB1_04_01	7984	3993	6.4	1292	1015	2304
HLA_DPA1_01_03_DPB1_04_02	4643	2322	0.4	824	634	1553
HLA_DPA1_01_03_DPB1_06_01	946	473	1.6	365	282	344
HLA_DPA1_01_03_DPB1_104_01	300	150	0	317	239	186
HLA_DPA1_02_01_DPB1_01_01	4292	2146	0.8	895	690	1353
HLA_DPA1_02_01_DPB1_09_01	2692	1346	0	686	528	890
HLA_DPA1_02_01_DPB1_10_01	3628	1814	2	842	657	1146
HLA_DPA1_02_01_DPB1_14_01	6035	3018	0.8	1245	972	1888
HLA_DPA1_02_01_DPB1_17_01	2170	1085	0	694	540	757
HLA_DPA1_02_01__DPB1_13_01	1968	984	0	704	543	696
HLA_DPA1_02_02_DPB1_05_01	7889	3945	4.8	1528	1221	2213
HLA_DQA1_01_01_DQB1_05_01	208	104	0	196	138	159
HLA_DQA1_01_02_DQB1_05_01	410	206	0	240	170	222
HLA_DQA1_01_02_DQB1_06_02	1498	749	8.8	412	306	545
HLA_DQA1_02_01_DQB1_02_02	5772	2886	12.4	955	744	1512
HLA_DQA1_02_01_DQB1_03_01	256	128	0	289	214	181
HLA_DQA1_03_01_DQB1_03_02	350	175	0	344	251	219
HLA_DQA1_03_02_DQB1_04_01	206	103	0	335	253	156
HLA_DQA1_05_01_DQB1_02_01	4051	2025	13.6	863	665	1138
HLA_DQA1_05_01_DQB1_03_01	617	307	28.4	440	332	308
HLA_DQA1_05_05_DQB1_03_01	5882	2941	9	1076	844	1620
HLA_DRB1_01_01	12412	6208	138.8	1686	1316	3987
HLA_DRB1_03_01	2178	1089	7.2	449	330	893
HLA_DRB1_04_01	5110	2557	11.2	874	670	1762
HLA_DRB1_04_02	256	128	0.8	250	184	186
HLA_DRB1_04_04	3076	1538	3.6	675	493	1026
HLA_DRB1_04_05	3972	1986	5.2	825	626	1301
HLA_DRB1_07_01	4466	2233	9.4	921	715	1549
HLA_DRB1_08_01	1118	559	0	503	372	425
HLA_DRB1_08_02	838	419	6	477	358	367
HLA_DRB1_09_01	1056	528	8	527	404	440
HLA_DRB1_10_01	2582	1291	1.6	745	586	880
HLA_DRB1_11_01	4180	2089	10.8	1005	792	1470
HLA_DRB1_11_03	422	211	0.8	509	390	221
HLA_DRB1_12_01	992	496	1.2	655	493	399
HLA_DRB1_13_01	1287	643	2.4	727	581	527
HLA_DRB1_13_02	1460	731	4.8	736	580	558
HLA_DRB1_13_03	1966	983	0	811	648	688
HLA_DRB1_14_01	681	340	0	643	517	286
HLA_DRB1_14_54	788	394	0	669	536	317
HLA_DRB1_15_01	4400	2201	11.2	1208	952	1518
HLA_DRB1_16_01	423	211	0.4	680	531	233
HLA_DRB3_01_01	1280	640	2.8	811	650	484
HLA_DRB3_02_02	1386	694	0.4	852	683	517
HLA_DRB3_03_01	210	105	0.4	702	574	160
HLA_DRB4_01_01	1316	658	4	977	760	521
HLA_DRB4_01_03	856	428	0.8	905	745	353
HLA_DRB5_01_01	3329	1664	8.8	1264	1016	1145
HLA_DRB5_02_02	926	463	0	922	753	366
Average	2646.37	1323.29	6.59	748.37	583.06	884.84

B.3 Binding motive

The binding motives are obtained by generating 200000 random 15-mer peptides and plotting, with the package *logomaker* [10], the core binders of the top 2000 highest predictions.
Below are the binding motives of the alleles retrieved from the IEDB website. In some cases, such as allele HLA_DRB3_03_01 or some of the DQ alleles, misalignments can be observed in the plots. For the prediction, all of the overlapping nonamers contained in the peptide are used, and the binding core is taken as the nonamer that contributes to the final prediction the most (as described in Appendix A.4). Therefore, the act of selecting a core is relevant for the logo plot but not for the prediction itself. In this regard, logo plots in CNN-PepPred involve a model reduction with loss of information. The missing weights in the logo from other overlapping nonamers contributing also to the binding score adds in this case to the common problems of low number of sequences and sequence bias in some peptide sets, rendering some of the logo plots, such as that for HLA_DRB3_03_01 (105 binding peptides) rather uninformative.


HLA_DPA1_01_03_DPB1_02_01
HLA_DPA1_01_03_DPB1_03_01
HLA_DPA1_01_03_DPB1_04_01
HLA_DPA1_01_03_DPB1_04_02
HLA_DPA1_01_03_DPB1_06_01
HLA_DPA1_01_03_DPB1_104_01
HLA_DPA1_02_01_DPB1_01_01
HLA_DPA1_02_01_DPB1_09_01
HLA_DPA1_02_01_DPB1_10_01
HLA_DPA1_02_01_DPB1_14_01
HLA_DPA1_02_01_DPB1_17_01
HLA_DPA1_02_01__DPB1_13_01
HLA_DPA1_02_02_DPB1_05_01
HLA_DQA1_01_01_DQB1_05_01
HLA_DQA1_01_02_DQB1_05_01
HLA_DQA1_01_02_DQB1_06_02
HLA_DQA1_02_01_DQB1_02_02
HLA_DQA1_02_01_DQB1_03_01
HLA_DQA1_03_01_DQB1_03_02
HLA_DQA1_03_02_DQB1_04_01
HLA_DQA1_05_01_DQB1_02_01
HLA_DQA1_05_01_DQB1_03_01
HLA_DQA1_05_05_DQB1_03_01
HLA_DRB1_01_01
HLA_DRB1_03_01
HLA_DRB1_04_01
HLA_DRB1_04_02
HLA_DRB1_04_04
HLA_DRB1_04_05
HLA_DRB1_07_01



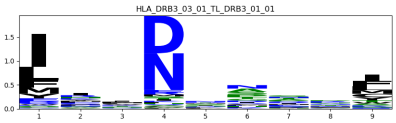
B.4 Transfer learning results

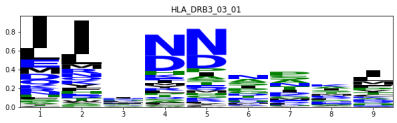
We used the trained IEDB models to perform another round of training using transfer learning (see Appendix A.5). The previously trained model of an allele was assigned to the training data of another one based on the similarity between their sequences. The same cross-validation set up was used to test the models. The new training instances, which contain nonamers present in the pre-trained model, were removed from the testing set but were kept for training. For this reason, the number of peptides tested in this cross-validation set up is lower than in Appendix B.2. For both optimization steps, the number of epochs was reduced to 15 (compared to 30 by default). The cross-validation results are reported in the table below. The "Allele" column contains first the name of the allele from which the training data

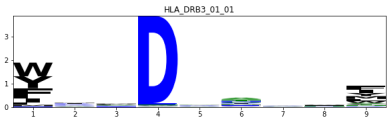
were taken, then the allele of the pre-trained model used for transfer learning. Both allele names are separated by "_TL_". The models with transfer learning systematically outperform the ones without. It is however possible that those performances do not generalize to fully new data due to the level of redundancy present in the IEDB data. In Appendix D, we can observe that, while the results on an independent evaluation set are slightly better with transfer learning, the increase in performance on the evaluation set is not proportional to the one in the cross-validation set up.

Allele	CNN-PepPred (transfer learning)						CNN-PepPred					
	#Peptide	#Binder	AUC	MCC	ACC	F1	#Peptide	#Binder	AUC	MCC	ACC	F1
HLA_DPA1_01_03_DPB1_02_01_TL_DPA1_01_03_DPB1_03_01	5025	2458	0.98	0.853	0.926	0.922	5177	2589	0.951	0.763	0.88	0.874
HLA_DPA1_01_03_DPB1_03_01_TL_DPA1_01_03_DPB1_02_01	4876	2372	0.982	0.867	0.933	0.931	5025	2512	0.959	0.803	0.901	0.898
HLA_DPA1_01_03_DPB1_04_01_TL_DPA1_01_03_DPB1_02_01	6764	3111	0.967	0.802	0.902	0.889	7984	3993	0.946	0.747	0.872	0.865
HLA_DPA1_01_03_DPB1_04_02_TL_DPA1_01_03_DPB1_02_01	3889	1578	0.986	0.881	0.943	0.929	4643	2322	0.971	0.839	0.919	0.917
HLA_DPA1_01_03_DPB1_06_01_TL_DPA1_01_03_DPB1_02_01	905	432	0.996	0.94	0.97	0.969	946	473	0.965	0.784	0.891	0.887
HLA_DPA1_01_03_DPB1_104_01_TL_DPA1_01_03_DPB1_02_01	288	138	0.998	0.952	0.976	0.975	300	150	0.99	0.887	0.943	0.944
HLA_DPA1_02_01_DPB1_01_01_TL_DPA1_01_03_DPB1_02_01	3806	1928	0.988	0.903	0.951	0.952	4292	2146	0.969	0.829	0.914	0.913
HLA_DPA1_02_01_DPB1_09_01_TL_DPA1_01_03_DPB1_02_01	2604	1261	0.991	0.922	0.961	0.96	2692	1346	0.972	0.835	0.917	0.916
HLA_DPA1_02_01_DPB1_10_01_TL_DPA1_01_03_DPB1_02_01	3527	1714	0.989	0.9	0.95	0.948	3628	1814	0.97	0.822	0.911	0.909
HLA_DPA1_02_01_DPB1_13_01_TL_DPA1_01_03_DPB1_02_01	1907	925	0.994	0.938	0.969	0.968	6035	3018	0.966	0.808	0.904	0.902
HLA_DPA1_02_01_DPB1_14_01_TL_DPA1_01_03_DPB1_02_01	5859	2846	0.982	0.865	0.932	0.93	2170	1085	0.974	0.839	0.919	0.918
HLA_DPA1_02_01_DPB1_17_01_TL_DPA1_01_03_DPB1_02_01	2094	1010	0.994	0.928	0.964	0.963	1968	984	0.975	0.845	0.922	0.919
HLA_DPA1_02_02_DPB1_05_01_TL_DPA1_01_03_DPB1_02_01	7629	3689	0.976	0.845	0.923	0.92	7889	3945	0.96	0.799	0.899	0.898
HLA_DQA1_01_01_DQB1_05_01_TL_DQA1_01_02_DQB1_06_02	112	72	0.999	0.924	0.964	0.972	208	104	0.94	0.741	0.87	0.867
HLA_DQA1_01_02_DQB1_05_01_TL_DQA1_01_02_DQB1_06_02	406	202	0.978	0.833	0.916	0.915	410	206	0.764	0.362	0.68	0.668
HLA_DQA1_01_02_DQB1_06_02_TL_DQA1_02_01_DQB1_02_02	1324	601	0.975	0.844	0.923	0.915	1498	749	0.915	0.672	0.835	0.829
HLA_DQA1_02_01_DQB1_02_02_TL_DQA1_01_02_DQB1_06_02	5576	2720	0.951	0.768	0.884	0.879	5772	2886	0.901	0.653	0.826	0.82
HLA_DQA1_02_01_DQB1_03_01_TL_DQA1_01_02_DQB1_06_02	256	128	0.986	0.875	0.938	0.938	256	128	0.884	0.603	0.801	0.794
HLA_DQA1_03_01_DQB1_03_02_TL_DQA1_01_02_DQB1_06_02	232	148	0.98	0.869	0.94	0.953	350	175	0.783	0.402	0.7	0.685
HLA_DQA1_03_02_DQB1_04_01_TL_DQA1_01_02_DQB1_06_02	189	102	0.962	0.831	0.915	0.92	206	103	0.792	0.488	0.743	0.728
HLA_DQA1_05_01_DQB1_02_01_TL_DQA1_01_02_DQB1_06_02	3661	1848	0.951	0.755	0.877	0.879	4051	2025	0.872	0.574	0.786	0.776
HLA_DQA1_05_01_DQB1_03_01_TL_DQA1_01_02_DQB1_06_02	336	105	0.975	0.815	0.92	0.873	617	307	0.909	0.668	0.833	0.825
HLA_DQA1_05_05_DQB1_03_01_TL_DQA1_01_02_DQB1_06_02	5656	2724	0.941	0.751	0.876	0.87	5882	2941	0.889	0.63	0.815	0.811
HLA_DRB1_01_01_TL_DRB1_15_01	10674	4935	0.877	0.586	0.794	0.765	12412	6208	0.824	0.492	0.744	0.73
HLA_DRB1_03_01_TL_DRB1_13_01	1887	956	0.961	0.792	0.895	0.893	2178	1089	0.866	0.553	0.775	0.763
HLA_DRB1_04_01_TL_DRB1_04_05	4231	2032	0.907	0.656	0.828	0.812	5110	2557	0.846	0.544	0.77	0.755
HLA_DRB1_04_02_TL_DRB1_04_04	67	54	0.94	0.676	0.896	0.935	256	128	0.764	0.469	0.734	0.73
HLA_DRB1_04_04_TL_DRB1_04_01	2234	1079	0.927	0.7	0.85	0.843	3076	1538	0.801	0.447	0.723	0.716
HLA_DRB1_04_05_TL_DRB1_04_01	3192	1524	0.966	0.809	0.905	0.898	3972	1986	0.913	0.676	0.837	0.83
HLA_DRB1_07_01_TL_DRB1_09_01	3665	1777	0.961	0.804	0.902	0.898	4466	2233	0.916	0.684	0.841	0.835
HLA_DRB1_08_01_TL_DRB1_13_03	1072	514	0.989	0.912	0.956	0.954	1118	559	0.96	0.827	0.913	0.911
HLA_DRB1_08_02_TL_DRB1_08_01	831	412	0.97	0.841	0.921	0.921	838	419	0.829	0.49	0.745	0.737
HLA_DRB1_09_01_TL_DRB1_07_01	306	113	0.966	0.823	0.915	0.89	1056	528	0.906	0.672	0.836	0.835
HLA_DRB1_10_01_TL_DRB1_01_01	2182	943	0.994	0.923	0.962	0.956	2582	1291	0.969	0.833	0.917	0.916
HLA_DRB1_11_01_TL_DRB1_13_03	4012	1945	0.945	0.736	0.868	0.861	4180	2089	0.917	0.665	0.832	0.826
HLA_DRB1_11_03_TL_DRB1_11_01	307	131	0.998	0.953	0.977	0.973	422	211	0.956	0.853	0.927	0.926
HLA_DRB1_12_01_TL_DRB1_13_01	950	470	0.991	0.907	0.954	0.953	992	496	0.966	0.809	0.904	0.903
HLA_DRB1_13_01_TL_DRB1_13_02	1145	558	0.982	0.876	0.938	0.936	1287	643	0.935	0.74	0.869	0.865
HLA_DRB1_13_02_TL_DRB1_13_01	1310	635	0.973	0.843	0.921	0.919	1460	731	0.885	0.607	0.803	0.802
HLA_DRB1_13_03_TL_DRB1_11_01	1823	844	0.995	0.942	0.971	0.969	1966	983	0.986	0.894	0.947	0.947
HLA_DRB1_14_01_TL_DRB1_13_03	668	327	0.997	0.958	0.979	0.979	681	340	0.988	0.918	0.959	0.959
HLA_DRB1_14_54_TL_DRB1_13_03	754	360	1	0.987	0.993	0.993	788	394	0.998	0.959	0.98	0.98
HLA_DRB1_15_01_TL_DRB1_01_01	2981	1350	0.96	0.79	0.896	0.885	4400	2201	0.887	0.615	0.806	0.796
HLA_DRB1_16_01_TL_DRB1_15_01	341	168	0.997	0.965	0.982	0.982	423	211	0.959	0.822	0.91	0.907
HLA_DRB3_01_01_TL_DRB3_02_02	915	521	0.966	0.836	0.916	0.922	1280	640	0.951	0.819	0.905	0.899
HLA_DRB3_02_02_TL_DRB3_01_01	941	587	0.996	0.941	0.972	0.978	1386	694	0.979	0.865	0.932	0.931
HLA_DRB3_03_01_TL_DRB3_01_01	188	83	0.996	0.968	0.984	0.982	210	105	0.963	0.803	0.9	0.896
HLA_DRB4_01_01_TL_DRB1_10_01	1226	595	0.98	0.879	0.94	0.938	1316	658	0.914	0.654	0.827	0.822
HLA_DRB4_01_03_TL_DRB4_01_01	678	251	0.998	0.949	0.976	0.968	856	428	0.971	0.824	0.911	0.908
HLA_DRB5_01_01_TL_DRB1_01_01	2222	927	0.966	0.834	0.919	0.904	3329	1664	0.913	0.676	0.837	0.833
HLA_DRB5_02_02_TL_DRB5_01_01	845	382	0.995	0.941	0.97	0.968	926	463	0.989	0.92	0.96	0.96
Average			0.975	0.857	0.93	0.926			0.921	0.716	0.857	0.853
Average weighted by #Binder			0.962	0.812	0.906	0.9			0.919	0.702	0.85	0.845

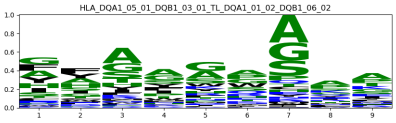
Transfer learning will also affect the logo plots. For example, as it can be seen in the figure below, using the pre-trained model of HLA_DRB3_01_01 has helped aligning the binding motif of the allele HLA_DRB3_03_01.

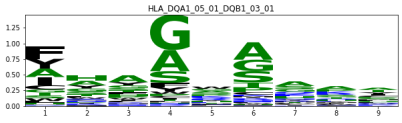


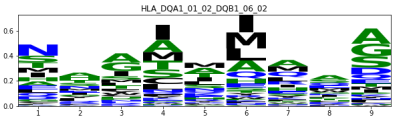




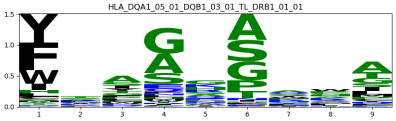
On the other hand, transfer learning with alleles that do not share strong fixed positions can reduce the alignment. For example, the pre-trained model of HLA_DQA1_01_02_DQB1_06_02 do not help aligning the binding motif of the allele HLA_DQA1_05_01_DQB1_03_01:

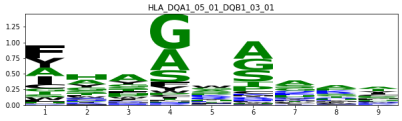


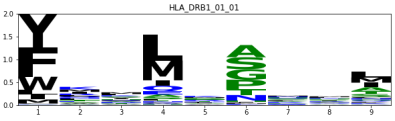




However, if instead the pre-trained model of the allele HLA_DRB1_01_01 is used, the shared patterns in position 1 and 6 help aligning the binding motif of the allele HLA_DQA1_05_01_DQB1_03_01:







Appendix C: NetMHCII data

In this appendix, we benchmark the convolutional neural network approach with the state-of-the-art methods from the netMHCII family [5]. The netMHCII family consists of two main methods: an allele-specific one (netMHCII) and a pan-specific one (netMHCIIpan). They are both based on the same core algorithm NNAlign ([7], [6]) which consists of a two-step optimization procedure that simultaneously estimates the core (nonamer) binder and the network weight configuration for the binding prediction. The pan-specific method is trained with all of the peptides of all of the alleles and can make predictions for all alleles with known alpha and beta chains. The pan-specific version is therefore more adequate for alleles with few training data. However, for alleles with enough training data, the authors report [5] that the allele specific method outperforms the pan specific one.

More modern versions of netMHCII include the possibility of training with multi-allele (MA) peptides [9]. While this is an interesting recent research direction ([1],[8],[2]), our aim in developing CNN-PepPred has been to provide an open-source efficient core algorithm that can be easily integrated in more complex pipelines and modified to fit specific purposes, including the incorporation of MA data.

The convolutional neural network approach is similar to the strategy of netMHCII, since both use similar blosum encoding and rely on an ensemble of neural networks. The main difference is that NNAlign is a two steps procedure that first identifies a core nonamer and applied it (with flanking region) to a network weight configuration. Our model uses convolution to slide through the possible nonamers contained within a peptide, therefore using the peptide in its full length. This strategy is also convenient to implement since it only requires building a sequential convolution neural network using user friendly libraries such as Keras.

NetMHCII methods are web based and meant to be used to predict binding with pre-trained models. While executables are available upon request, the core algorithm training the models (NNAlign) is not open-source and full development details have not been, to our knowledge, provided in any publication. As CNN-PepPred, NNAlign can be also used as an executable to train models with specific data sets.

C.1 Cross-validation result

The data and the 5 fold cross-validation partition for this set-up were taken from the paper presenting netMHCIIpan-3.2, which is the latest version of the model not including multiple-allele data. The results of netMHCIIpan-3.2 and netMHCII-2.3 were taken from the supplementary file, Suppl Table 3, of [5]. The authors only reported the AUC score but we also included the Pearson correlation (PC) and root mean squared error (RMSE) scores of our model for further information. The best AUC score for each allele is highlighted in bold.

We used CNN-PepPred with default parameters, namely, an ensemble of 10 nets per number of filters (5/10/20/30), totalling 40 nets. The threshold used to binarize the quantitative outcome was set to the log50k transform of 500nM as in [5], namely $1 - \log(500)/\log(50000) \approx 0.426$.

As it can be seen in the table, if we also include the alleles with few training data (for which allele-specific methods are clearly not fitted), netMHCIIpan outperforms (on average) the two allele specific methods. However, considering different sets of alleles with different minimum numbers of binding training peptides, our model outperforms (on average) the models from the netMHCII family. In any cases, the performances are overall similar.

Allele	#Peptide	#Binder	CNN-PepPred			NetMHCII-2.3	NetMHCIIPan-3.2
			PC	AUC	RMSE	AUC	AUC
DRB1_0101	10412	6376	0.69	**0.837**	0.195	0.829	0.832
DRB1_0103	42	4	-0.231	0.204	0.208	0.25	**0.678**
DRB1_0301	5352	1457	0.646	**0.836**	0.181	0.816	0.816
DRB1_0401	6317	3022	0.613	**0.811**	0.198	0.798	0.809
DRB1_0402	53	19	0.419	0.669	0.249	0.633	**0.701**
DRB1_0403	59	14	0.511	0.703	0.152	0.644	**0.841**
DRB1_0404	3657	1852	0.636	0.803	0.189	0.787	**0.812**
DRB1_0405	3962	1653	0.669	**0.841**	0.171	0.839	0.827
DRB1_0701	6325	3456	0.748	**0.884**	0.171	0.877	0.875
DRB1_0801	937	390	0.658	0.836	0.165	0.834	**0.844**
DRB1_0802	4465	2036	0.673	**0.838**	0.184	0.834	0.834
DRB1_0901	4318	2164	0.657	**0.833**	0.175	0.832	**0.833**
DRB1_1001	2066	1521	0.754	0.915	0.157	0.912	**0.923**
DRB1_1101	6045	2667	0.734	0.866	0.174	**0.867**	0.864
DRB1_1201	2384	759	0.771	**0.894**	0.141	0.891	0.868
DRB1_1301	1034	520	0.673	0.851	0.22	0.828	**0.857**
DRB1_1302	4477	2249	0.774	**0.89**	0.176	0.889	0.885
DRB1_1501	4850	2107	0.679	**0.839**	0.187	0.833	0.834
DRB1_1602	1699	989	0.778	**0.886**	0.151	0.879	0.883
DRB3_0101	4633	1415	0.813	**0.912**	0.149	0.898	0.888
DRB3_0202	3334	1055	0.808	**0.889**	0.171	0.887	0.869
DRB3_0301	884	510	0.646	0.826	0.192	0.824	**0.84**
DRB4_0101	3961	1540	0.706	**0.851**	0.171	0.837	0.822
DRB4_0103	846	525	0.67	**0.849**	0.197	0.839	0.841
DRB5_0101	5125	2430	0.714	**0.855**	0.191	0.849	0.849
H_2_IAb	1794	431	0.703	0.885	0.163	0.884	**0.894**
H_2_IAd	774	321	0.611	0.813	0.202	**0.819**	**0.819**
H_2_IAk	115	4	0.332	0.619	0.137	0.628	**0.635**
H_2_IAs	190	48	0.534	0.815	0.195	0.761	**0.825**
H_2_IAu	56	22	0.603	**0.898**	0.262	0.83	0.765
H_2_IEd	245	28	0.4	0.706	0.18	0.73	**0.754**
H_2_IEk	68	40	0.633	0.754	0.216	0.836	**0.853**
HLA_DPA10103_DPB10201	787	141	0.72	0.903	0.146	0.91	**0.917**
HLA_DPA10103_DPB10301	1563	575	0.796	**0.914**	0.166	**0.914**	0.902
HLA_DPA10103_DPB10401	2725	786	0.882	**0.939**	0.14	0.935	0.935
HLA_DPA10103_DPB10402	45	9	0.194	0.596	0.18	0.497	**0.71**
HLA_DPA10103_DPB10601	584	282	0.958	0.995	0.116	**0.996**	0.995
HLA_DPA10201_DPB10101	2447	859	0.833	0.897	0.149	**0.903**	**0.903**
HLA_DPA10201_DPB10501	2470	713	0.806	0.913	0.154	**0.914**	0.911
HLA_DPA10201_DPB11401	2302	849	0.851	**0.942**	0.151	0.937	0.93
HLA_DPA10301_DPB10402	2641	921	0.834	0.903	0.157	**0.906**	0.904
HLA_DQA10101_DQB10501	2946	815	0.813	**0.917**	0.138	**0.917**	0.9
HLA_DQA10102_DQB10501	833	458	0.662	0.865	0.194	**0.867**	0.839
HLA_DQA10102_DQB10502	800	158	0.675	**0.851**	0.159	0.85	0.835
HLA_DQA10102_DQB10602	2747	1256	0.814	0.902	0.148	**0.905**	0.89
HLA_DQA10103_DQB10603	462	90	0.503	0.803	0.199	0.816	**0.861**
HLA_DQA10104_DQB10503	883	105	0.635	0.837	0.143	**0.844**	0.805
HLA_DQA10201_DQB10202	944	119	0.644	**0.86**	0.131	0.851	0.814
HLA_DQA10201_DQB10301	827	374	0.696	**0.876**	0.187	0.864	0.849
HLA_DQA10201_DQB10303	761	265	0.721	0.886	0.152	0.887	**0.894**
HLA_DQA10201_DQB10402	768	241	0.638	0.854	0.181	0.858	**0.86**
HLA_DQA10301_DQB10301	207	66	0.591	0.774	0.195	0.761	**0.839**
HLA_DQA10301_DQB10302	3111	568	0.702	0.846	0.126	**0.849**	0.81
HLA_DQA10303_DQB10402	567	117	0.632	**0.844**	0.168	0.836	0.82
HLA_DQA10401_DQB10402	2890	928	0.794	**0.903**	0.116	0.894	0.883
HLA_DQA10501_DQB10201	2897	874	0.78	**0.889**	0.131	**0.889**	0.876
HLA_DQA10501_DQB10301	3585	1812	0.812	**0.926**	0.143	0.922	0.915
HLA_DQA10501_DQB10302	847	203	0.6	0.82	0.139	**0.831**	0.822
HLA_DQA10501_DQB10303	564	179	0.68	0.869	0.138	**0.884**	0.876
HLA_DQA10501_DQB10402	749	337	0.718	**0.877**	0.157	0.857	0.868
HLA_DQA10601_DQB10402	565	133	0.622	**0.854**	0.18	0.845	0.848
Average			0.666	0.839	0.17	0.833	**0.847**
Average weighted by #Binder			0.722	**0.865**	0.172	0.86	0.858
Average over alleles with >=100binders			0.723	**0.872**	0.164	0.869	0.864
Average over alleles with >=500binders			0.745	**0.876**	0.165	0.871	0.867
Average over alleles with >=1000binders			0.719	**0.863**	0.174	0.856	0.854

C.2 Run time comparison

To evaluate computational time, we used different numbers of sequences of the proteome of *Burkholderia pseudomallei* (`https://www.uniprot.org/proteomes/UP000000605`) to the trained model of HLA-DRB1*07:01 (with the data set from NetMHCIIpan3.2). Each sequence was cut into all of its overlapping 15-mers and a prediction was made for all of them. In the table below, the time is reported in seconds for different number of sequences. The full proteome contains 5717 sequences corresponding to 1908278 15-mers. Note that the number of 15-mers corresponds to the non-unique amount (no check performed for sequence identity). The last row of the table corresponds to the application of the model against the human proteome with ca. 75000 proteins (`https://www.uniprot.org/proteomes/UP000005640`); it was added to have an idea of how long the GPU version would take on really big data sets (more than 20 millions non-unique 15-mers). We tested our model using the GPU and CPU versions and we downloaded the latest version of netMHCIIpan4.0 from its web-server (`https://services.healthtech.dtu.dk/service.php?NetMHCIIpan-4.0`) and ran it for allele DRB1*07:01, with length 15 and the print unique binding core option. We used NNAlign with the same parameters as in Appendix B.2 (except for output rescaling which was set to the default, since the training outcome is quantitative).
Results can be found in the table below. For the application of a trained model on new instances, the fastest model is NNAlign followed by CNN-PepPred with GPU.
We can also notice that the time grows more or less linearly with increasing number of sequences, which makes sense as our model analyses new sequences in batches of 50000 (by default). The application in batches also means that there will never be a memory issue whatever the number of peptides to analyse. For the GPU computation we used NVIDIA GeForce GTX 1080 under Windows OS. The CPU runs were performed for CNN-PepPred, NNAlign and netMHCIIpan under Windows Subsystem for Linux (WSL) in the same computer (equipped with an AMD Ryzen 7 1700 8-core), since the executables of the latter two require a Linux OS. We couldn't perform the GPU computations on WSL since the system does not support it.

#seq	#15-mers (non-unique)	time[s]			
		CNN-PepPred (cpu)	CNN-PepPred (gpu)	NetMHCIIpan4.0	NNAlign2.1
100	33130	41	27	239	6
500	157401	153	71	1121	12
1000	333632	323	127	2369	24
2000	666696	630	237	4738	44
3000	1006260	953	355	7118	68
4000	1336574	1256	470	9412	89
5000	1663243	1530	581	11754	108
5717	1908278	1774	664	13466	127
75069	24752058		4184		738

Appendix D: Evaluation set

For the evaluation of the models trained with the data retrieved from IEDB as described in Appendix B.1, we used the T-cell epitope benchmark from Jensen et al.([5]). This data set contains, for different alleles, several pairs of epitope and epitope-source protein sequences. The epitope source is split into all of its overlapping l-mers, where l is the length of the epitope. The actual epitope is labelled as binding while the overlapping non-epitope l-mers in the epitope source are labelled as non-binding. The trained model is then applied to all l-mers and the evaluation is performed using two metrics: the AUC and the F-rank. The F-rank corresponds to the ratio between the number of peptides from the source with predicted binding score higher than that of the epitope and the total number of peptides in the source. Therefore, a value of 0 means that no peptides are predicted as stronger binders than the epitope and a value of 1 means that all peptides are predicted as stronger binders than the epitope. Both scores are computed for each pair separately and averaged per allele. As noted by the authors, this evaluation will tend to underestimate the performances since some negatively labelled peptides might still be presented by the human MHC molecule.
We selected the epitopes for alleles that are present in our data set (listed in section 2.5). We then removed a few epitopes that were already present in our training data set. The table below contains the scores of this evaluation. The results for CNN-PepPred are overall similar to the ones of NetMHCIIpan3.2, with a slight advantage on average for CNN-PepPred: the average F-rank/AUC is 0.174/0.825 for CNN-PepPred and 0.193/0.806 for NetMHCIIpan3.2 (with the values as reported by the authors in suppl. table 5 of the paper's supplementary file).

Allele	CNN-PepPred			NetMHCIIpan3.2		
	#Epitope	average F-rank	average AUC	#Epitope	average F-rank	average AUC
HLA_DPA1_01_03_DPB1_02_01	1	**0.005**	**0.995**	1	0.02	0.98
HLA_DQA1_01_02_DQB1_06_02	2	0.085	0.915	2	**0.051**	**0.948**
HLA_DRB1_01_01	235	0.194	0.806	240	**0.181**	**0.818**
HLA_DRB1_03_01	96	0.147	0.853	101	**0.14**	**0.86**
HLA_DRB1_04_01	220	**0.157**	**0.842**	232	0.195	0.804
HLA_DRB1_04_02	3	0.22	0.78	3	**0.206**	**0.793**
HLA_DRB1_04_04	142	0.266	0.734	146	**0.19**	**0.81**
HLA_DRB1_04_05	3	**0.01**	**0.99**	3	0.03	0.964
HLA_DRB1_07_01	196	**0.159**	**0.841**	197	0.179	0.821
HLA_DRB1_08_01	19	**0.199**	**0.801**	22	0.24	0.76
HLA_DRB1_09_01	40	**0.277**	**0.721**	40	0.326	0.672
HLA_DRB1_10_01	9	0.496	0.503	10	**0.328**	**0.672**
HLA_DRB1_11_01	192	**0.106**	**0.894**	196	0.14	0.859
HLA_DRB1_12_01	2	0.139	0.86	2	**0.086**	**0.914**
HLA_DRB1_13_01	12	**0.087**	**0.913**	12	0.245	0.754
HLA_DRB1_13_02	3	**0.293**	**0.706**	3	0.547	0.45
HLA_DRB1_14_01	20	0.234	0.766	20	**0.206**	**0.795**
HLA_DRB1_15_01	113	**0.18**	**0.82**	122	0.184	0.815
HLA_DRB3_01_01	4	**0.034**	**0.966**	4	0.068	0.932
HLA_DRB3_02_02	7	**0.144**	**0.856**	7	0.149	0.85
HLA_DRB4_01_01	3	**0.279**	**0.72**	3	0.372	0.628
HLA_DRB5_01_01	119	**0.123**	**0.877**	120	0.17	0.83
Average		**0.174**	**0.825**		0.193	0.806

This evaluation set up was also performed on the models trained with transfer learning (see Appendix A.5 and Appendix B.4). As discussed in Appendix B.4, while the results are on average slightly better with transfer learning, the increment in predictive performance on the evaluation set does not match the one in the cross-validation set up. This might be due to the redundancy present in the IEDB data. The previously trained models are already adapted to this type of data and using transfer learning helps finding patterns, however it doesn't seem to generalize as well with independent data. Therefore, some caution must be observed when using transfer learning to train and the parameters should be set to reduce the learning during optimization (lower learning rate and/or number of epochs).

Allele	#Epitope	CNN-PepPred		CNN-PepPred (transfer learning)	
		average F-rank	average AUC	average F-rank	average AUC
HLA_DPA1_01_03_DPB1_02_01	1	0.005	0.995	**0.004**	**0.996**
HLA_DQA1_01_02_DQB1_06_02	2	**0.085**	**0.915**	0.196	0.803
HLA_DRB1_01_01	235	**0.194**	**0.806**	**0.194**	**0.806**
HLA_DRB1_03_01	96	**0.147**	**0.853**	0.151	0.849
HLA_DRB1_04_01	220	0.157	0.842	**0.149**	**0.851**
HLA_DRB1_04_02	3	**0.22**	**0.78**	0.294	0.705
HLA_DRB1_04_04	142	**0.266**	**0.734**	0.28	0.719
HLA_DRB1_04_05	3	0.01	0.99	**0.009**	**0.991**
HLA_DRB1_07_01	196	**0.159**	**0.841**	0.165	0.835
HLA_DRB1_08_01	19	**0.199**	**0.801**	0.203	0.797
HLA_DRB1_09_01	40	0.277	0.721	**0.257**	**0.742**
HLA_DRB1_10_01	9	0.496	0.503	**0.45**	**0.55**
HLA_DRB1_11_01	192	**0.106**	**0.894**	0.111	0.889
HLA_DRB1_12_01	2	0.139	0.86	**0.028**	**0.972**
HLA_DRB1_13_01	12	0.087	0.913	**0.084**	**0.916**
HLA_DRB1_13_02	3	0.293	0.706	**0.221**	**0.779**
HLA_DRB1_14_01	20	**0.234**	**0.766**	0.246	0.753
HLA_DRB1_15_01	113	0.18	0.82	**0.168**	**0.832**
HLA_DRB3_01_01	4	0.034	0.966	**0.032**	**0.968**
HLA_DRB3_02_02	7	0.144	0.856	**0.129**	**0.87**
HLA_DRB4_01_01	3	0.279	0.72	**0.227**	**0.773**
HLA_DRB5_01_01	119	**0.123**	**0.877**	0.15	0.85
Average		0.174	0.825	**0.17**	**0.829**

References

[1] B. Alvarez, B. Reynisson, C. Barra, S. Buus, N. Ternette, T. Connelley, M. Andreatta, and M. Nielsen. NNAlign_MA; MHC Peptidome Deconvolution for Accurate MHC Binding Motif Characterization and Improved T-cell Epitope Predictions. *Molecular & Cellular Proteomics*, 18(12):2459–2477, 2019.

[2] B. Chen, M. S. Khodadoust, N. Olsson, L. E. Wagar, E. Fast, C. L. Liu, Y. Muftuoglu, B. J. Sworder, M. Diehn, R. Levy, M. M. Davis, J. E. Elias, R. B. Altman, and A. A. Alizadeh. Predicting HLA class II antigen presentation through integrated deep learning. *Nature Biotechnology*, 37(11):1332–1343, 2019.

[3] F. Chollet et al. Keras, 2015.

[4] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the USA*, 89(22):10915–10919, 1992.

[5] K. K. Jensen, M. Andreatta, P. Marcatili, S. Buus, J. A. Greenbaum, Z. Yan, A. Sette, B. Peters, and M. Nielsen. Improved methods for predicting peptide binding affinity to MHC class II molecules. *Immunology*, 154(3):394–406, 2018.

[6] M. Nielsen and M. Andreatta. NNAlign: a platform to construct and evaluate artificial neural network models of receptor-ligand interactions. *Nucleic Acids Research*, 45(W1):W344–W349, 2017.

[7] M. Nielsen and O. Lund. NN-align. An artificial neural network-based alignment algorithm for MHC class II peptide binding prediction. *BMC Bioinformatics*, 10:296, 2009.

[8] J. Racle, J. Michaux, G. A. Rockinger, M. Arnaud, S. Bobisse, C. Chong, P. Guillaume, G. Coukos, A. Harari, C. Jandus, M. Bassani-Sternberg, and D. Gfeller. Robust prediction of HLA class II epitopes by deep motif deconvolution of immunopeptidomes. *Nature Biotechnology*, 37(11):1283–1286, 2019.

[9] B. Reynisson, B. Alvarez, S. Paul, B. Peters, and M. Nielsen. NetMHCpan-4.1 and NetMHCIIpan-4.0: improved predictions of MHC

antigen presentation by concurrent motif deconvolution and integration of MS MHC eluted ligand data. *Nucleic Acids Research*, 48(W1):W449–W454, 2020.

[10] A. Tareen and J. B. Kinney. Logomaker: beautiful sequence logos in Python. *Bioinformatics*, 36(7):2272–2274, 2019.