Predicting chromatin accessibility using neural networks

Max Gubert Olivé

Project in Bioinformatics

Bioinformatics Research Center

Aarhus University

Denmark

January 2022

Contents

Introduction

The use of neural networks on genomic data is not something new. Given training data, models parametrized by machine learning can effectively predict protein binding, DNA accessibility, histone modifications, and an assort of other things. Convolutional neural networks adaptively learn from the data features during training and then apply nonlinear transformations to map input data to informative high-dimensional representations that trivialize classification or regression [1].

In this project, we aim to build a neural network that can effectively predict the signal enrichment of DNase I hypersensitivite sites in a specific cell line, compared to a simulated control distribution. In other words, we want to predict how open or closed chromatin regions are and thus identify active genomic sites. The model will be built with information on how DNase hypersensitivity varies, using DNase-seq tracks and signal values from the Encyclopedia of DNA Elements (ENCODE) project [2]. ENCODE is an international collaboration of research groups founded by the National Human Genome Research Institute, and its goal is to build a comprehensive list of the functional elements in the human genome. The assays needed to achieve this goal can be very expensive and technical challenges can prove a problem. Because of this, computational methods capable of predicting the outcome of these assays are of great importance.

DNase I hypersensitivite sites (DHSs) are regions of chromatin that are sensitive to being cleaved by the DNase I enzyme. This is due to the chromatin being less condensed in these regions, which exposes the DNA and makes it accessible for this enzime to act upon it. Of course, this is related to the transcriptional activity as well because transcription can only happen when the chromatin is in its "open state" and it has enough space to allow the binding of transcription factors. Many regulatory elements in DNA regions modify this accessibility in order to regulate transcription. Mutations in non-coding regions could also modify chromatin accessibility as well. Using the data provided by ENCODE, our neural networks will learn the relevant sequence motifs and the regulatory logic needed to determine DNA accessibility on genomic sequences.

The structure of a fully connected neural network

A neural network is a highly parametrized model inspired by the architecture of the human brain. It contains memory units called *neurons* which automatically learn new features from the data through a process called *supervised learning*, in which the building of a function that maps an input to an output is based on example input-output pairs. Each of the neurons is connected to the input layer via a vector of parameters called *weights*, which indicates how much the response varies, or in other words it tells the neural network how much importance is given to an input. In these types of network, several neurons located in a *hidden layer* take the inputs and generate an output, which is the weighted sum of all the inputs. All the outputs are then combined in an output layer to generate predictions, and lastly the error of the prediction is computed by comparing the output with the real known value. Then the weights are updated accordingly to the error measured. This is done over and over in an attempt to improve the predictions, and each step is called an *epoch*. Each row of neurons is called a hidden layer, and the output of a neuron can be fed to another neuron thus creating multiple hidden layers (Fig. 1). [3] [4]

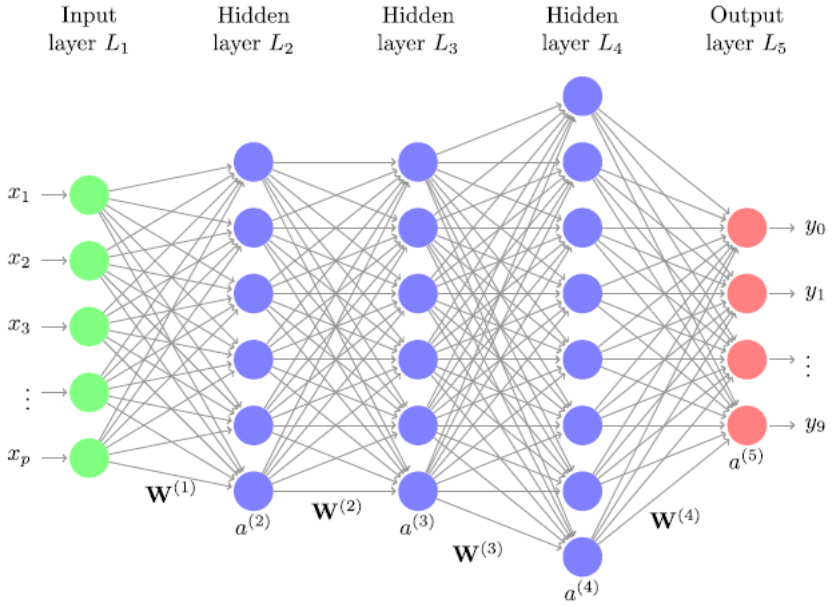

Figure 1: Example of a neural network with 5 layers: 1 input layer, 3 hidden layers, 1 output layer. Notice that multiple outputs are also possible. [3]

It is often the case that the weighted sum of all the inputs is passed through an *activation function* to produce an output. This functions is usually nonlinear, and it takes the result

of the neuron and transforms it to be fed to the next layer. Nonlinear transformations are applied hopping that the transformed data will be close to linear for regression. In all our neural networks we used the Rectified Linear Unit (ReLU) function:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

which transforms all negative values to zero (Fig. 2). We use this activation function because it improves performance and makes the gradient computation simple.

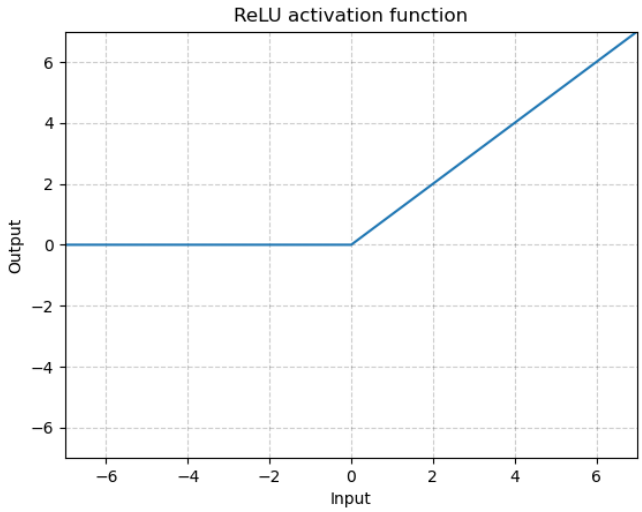

Figure 2: Visualization of the ReLU function. [5]

Gradient descent

With the basic structure of a neural network being defined, the training of the model is commonly done using *gradient descent* and *backpropagation*. Given a generic pair (input-output), we first make a forward pass through the network which activates each of the nodes in each of the layers including the output layer, and produces a prediction. An error is then computed between the prediction and the actual output. The weights are updated accordingly by taking the current weight and subtracting the partial derivative of the error with respect to the weight. Each weight measures the responsibility of each node for the error in predicting the true output. This process is known as *fitting* the neural network. One of the most common algorithms that utilizes this idea is *mini-batch stochastic gradient*

descent, which takes random starting values for the weights (except zero). In this method, small batches of data are sampled at random, which increases the efficiency. The algorithm computes the gradient or derivative of the loss function, which is the rate of change of the function. It is a vector (a direction of move) that points in the direction of the greatest increase of a function, which is zero at a local maximum or local minimum (because there is no single direction of increase). An epoch is completed after the neural network has gone through all the data and updated the weights, and all training samples have been used in the gradient steps, independently of how they have been grouped or how many different gradient steps have been made. Neural networks are trained over many epochs to ensure optimal performance.

Convolutional neural networks

A *convolutional neural network* is a class of neural network with a more complex architecture. This type of neural network adaptively learns features from the data during training using what are called *filters* or *kernels*. The convolutions are useful to preserve locality, i.e., information of the nearby features in the input. Nonlinear transformations such as ReLU are also applied to map the input data to informative high-dimensional representations that trivialize classification or regression. Convolutional neural networks might not require much data pre-processing because the filters will be learned by the network itself.

The convolutional layer is often the first layer of the network. It can be followed by additional convolutional layers, pooling layers and hidden layers. With each layer the convolutional neural network increases its complexity, and the more layers, the more details of the data it can map and recognize potentially. The convolutional layer is the core building block of the convolutional neural network. It requires a few components, which are input data, a filter or kernel, and a feature map. The filter will move across the receptive fields of the data, checking if the feature is present. This process is known as a convolution.

The filter is an array of weights which is applied to the data to calculate a dot product between the each data point and the corresponding weight, which is then fed to an output array. Once this is done the filter moves by a stride and repeats the process until the kernel has swept across the entire image (Fig. 3). Each output value in the feature map does not

have to connect to each pixel value in the input image. It only needs to connect to the receptive field, where the filter is being applied. Since the output array does not need to map directly to each input value, convolutional (and pooling) layers are commonly referred to as "partially connected" layers. The final output is what's called a *feature or activation map* [6].

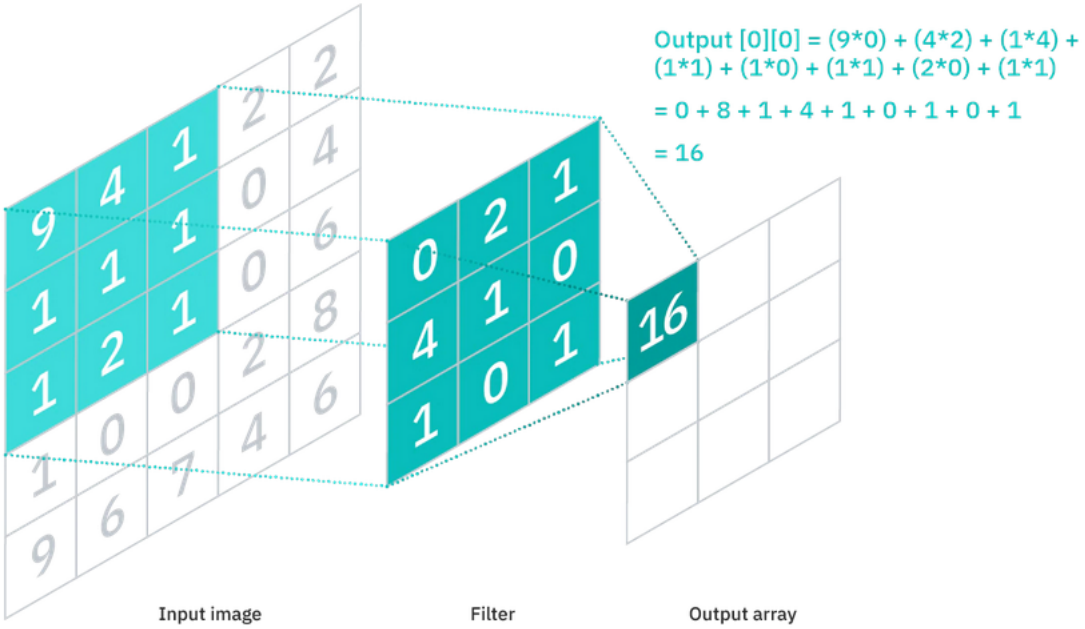

Figure 3: Convolutional layer, where a 3x3 filter is being applied to the input image to produce the output array [6]

An additional layer that can be added to the network is a pooling layer, which reduces the dimension of each non-overlapping block by usually computing their maximum. The pooling filter is often a 2x2 matrix which moves through the layer by a stride the same way as a convolution filter. It reduces the amount of parameters and computation in the network and helps reducing overfitting by compressing or generalizing the features in the feature map.
A final fully connected layer with hidden layers can be added after the convolution and pooling layer to perform the task of classification or regression based on the features extracted through the previous layers and their different filters. A final activation function such as ReLU is used to produce the output. The structure of a complete convolutional neural network is shown in Figure 4.

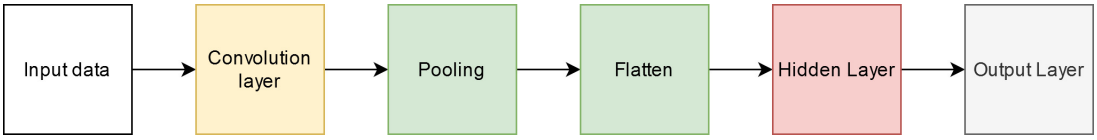

Figure 4: Structure of our convolutional neural network.

Neural network hyperparameters

Gradient descent is used to tune the weights that are applied to the input data, but there are other parameters that influence how well a neural network will perform. For instance, one of these hyperparameters is the number of hidden layers or the amount of units in the hidden layer. More hidden layers and more units mean more complexity of the model, which results in increased computation time.
We designed two fully connected neural networks, one with one hidden layer and another with two hidden layers, and we designed all our convolutional neural networks with one convolutional, one pooling and one hidden layer. We experimented with different hyperparameters using an empirical approach, meaning that we tried different values and chose the one that resulted in the best model performance.

In this project, we first experimented with different architectures for both fully connected and convolutional neural networks. We then optimized the hyperparameters of the neural networks in order to increase the performance of the models in predicting chromatin accessibility. Finally, we selected the best neural network and predicted the signal enrichment values in promoter vs non-promoter regions of the genome. In depth details on how these neural networks were implemented, as well as details on how the data was prepared and loaded, are explained in the following section.

Materials and Methods

Data

The human genome sequence (release version hg38/GRCh38) and the genome-wide signal tracks were downloaded from the ENCODE Project Consortium. The Python libraries *py2bit* and *pyBigWig* were used to handle the reference genome and tracks, respectively. We used chromosome 22 and the C02 adrenal gland (M22 H3K4me3) track for our experiments. The specific DNase-seq measurements in the tracks contained the statistical significance of enrichment (-$\log_{10}$ p-value) at each genomic position compared to the expected number of reads from a simulated Poisson distribution. Genomic regions with a significant signal enrichment are presumed to be regulatory elements.

Data pre-processing

Prior to model design and implementation, the data was pre-processed into a proper format for our learning algorithms to ensure good performance. The original format of the reference genome was converted from a string of DNA characters (A, C, G, T) into binary vectors, where each character was one-hot encoded as a vector of zeros except of a one in a specific position. A was encoded as (1, 0, 0, 0), C as (0, 1, 0, 0), G as (0, 0, 1, 0) and T as (0, 0, 0, 1).

The data was divided into windows of a specified size (201 positions by default) and the mean enrichment value was calculated for each window. Windows with unidentified nucleotides (defined by the letter N) and with missing values were ignored for further analysis. From a total of 252828 windows of size 201, 58054 were discarded due to unidentified nucleotides in the sequence and none was found with missing values, leaving a total of 194774 windows left. The reason for not finding any missing values was because the N's were filtered first, removing all the missing values present in the dataset. By first looking at missing values, we found 50 windows to remove, with the rest of the windows being removed afterwards due to unidentified nucleotides. The files were then converted into BED (Browser Extensible Data) format, containing the chromosome name, the window start and end, the one-hot encoding of the sequence and the mean enrichment values.

In order to avoid hard-to-map regions, the data was filtered using mappability tracks obtained using Umap and Bismap [7]. A single-read mappability file containing regions with at least one alignable k-mer of length k=100 was intersected with our data using *pybedtools* [8] [9], and the windows with at least one overlap were kept. A total of 24120 windows were skipped due to hard-to-map regions, and the remaining 170654 windows were used for our experiments.

To reduce the effect of outliers in the DNA-seq measurements, we applied the hyperbolic arcsine (arcsinh) transformation to the values [10], and used the arcsinh-transformed signal

$$sinh^{-1}x = ln\left(x + \sqrt{1 + x^2}\right)$$

in all models implemented.

The data was then split into three random partitions for training (90%), validation (5%) and testing (5%). The training set was used for model and hyperparameter optimization, the validation set for *early stopping* and the final model performance was assessed using the test set.

PyTorch and neural network implementation

The open source machine learning Python library *PyTorch* [11] was used for implementing our neural networks. PyTorch uses Tensors to store and operate on large multidimensional arrays of numbers using graphic processing units (GPUs) for accelerated computation, and provides an useful automatic differentiation system effective for neural network implementation.

The data was converted into tensors to serve as input for the neural networks, and was loaded in batches to speed up data loading. After the first hidden/convolutional layer, batch normalization was applied to the layer to standardize the mean to zero and the standard deviation to one. We applied dropout (p=0.25) in all our neural networks prior to the output layer. Dropout is a technique that prevents overfitting by inactivating random neurons before the final output layer. All our models were trained using the ADAM optimizer due to its extensive use and proven success [12], and we used the mean squared error (MSE) as loss function. The training was done over 50 epochs or until early stopping was triggered. This function stopped the training of the model if the validation loss did not

decrease after 3 consecutive epochs, in order to reduce the chance of overfitting. We experimented with different hyperparameters in both our fully connected and convolutional neural networks, and the optimization outcomes are reported in the results section. We conducted each experiment twice in order to get a better estimation of how the model performs.

Signal enrichment prediction

The best performance neural network was selected to make predictions on promoter vs non-promoter regions of chromosome 22. Windows of size 201 that were inside a promoter region were identified by intersecting them with a BED file containing the positions of all promoters in the chromosome. The test set was used to make the predictions, and *pybedtools* to identify windows inside a promoter.

GenomeDK

GenomeDK [13] is a high performance computing facility located at Aarhus University, Denmark, designed to assist bioinformatics and life science research by providing large data storage, high-speed data processing and strong computing power for complex calculations. We used GenomeDK for all our experiments, including data pre-processing and neural network training and optimization, and took advantage of the cluster's GPUs to speed up processing-intensive operations.

Results

Data exploration

Before starting to build learning algorithms, we decided to explore and visualize the data in order to understand the characteristics of the dataset. After pre-processing the data, eliminating windows with unidentified nucleotides and missing values, and ignoring hard-to-map regions, we computed the summary statistics of the data (Table 1). In order to reduce the effect of outliers, we transformed the signal enrichment values using the hyperbolic arcsine (arcsinh) function and plotted the distribution of the data after being transformed (Fig. 5).

Dataset	Windows	Mean	Median	SD	Min	Max
Non-transformed	170654	0.598	0.139	2.296	0	338.166
Arcsinh transformed	170654	0.371	0.139	0.560	0	6.517

Table 1: Descriptive statistics of the dataset.

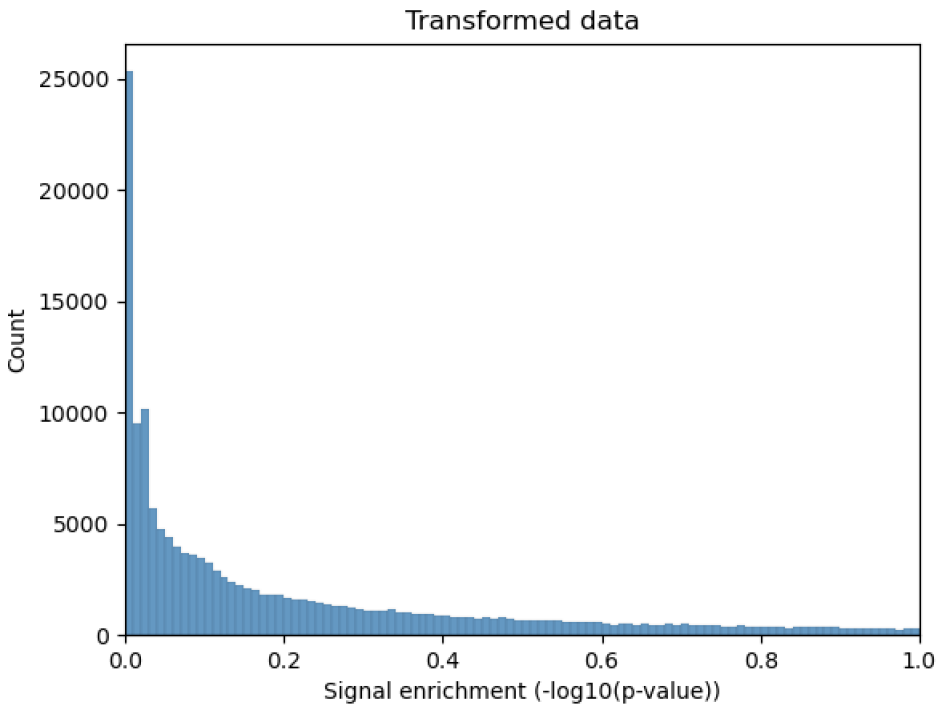

Figure 5: Transformed data distribution. The x-axis is cut at 1.0 for better visualization.

Baseline models

A baseline model is a simple model that can provide reasonable results, is less likely to overfit, is usually quick and requires low computational cost. We used the training and the test dataset and implemented several baseline models using the *scikit-learn* package [14] to get a better understanding of our data and to have something to compare with our neural networks. We tested our dataset with two dummy regressors, one that always predicts the mean and another one that predicts the median. The MSE of the mean and median dummy regressors was 0.315 and 0.370, respectively. Linear regression predicted enrichment values with a MSE of 0.290, and when we tested Lasso regression with different α values, the best MSE was 0.290 with $\alpha = 0$, so the same coefficients as linear regression. Ridge regression's best MSE was 0.290 with $\alpha = 0.95$. We also built a more complex model using random forest with random parameters, obtaining a MSE of 0.066. An overview of the baseline models implemented and their performance can be found in Table 2.

Model	MSE
Dummy Regressor Mean	0.315
Dummy Regressor Median	0.370
Linear Regression	0.290
Lasso Regression ($\alpha = 0$)	0.290
Ridge Regression ($\alpha = 0.95$)	0.290
Random Forest	0.066

Table 2: Baseline models performance.

Fully connected neural network and hyperparameter optimization

We implemented our first fully connected neural network (1 input layer, 1 hidden layer and 1 output layer) with random parameters (1000 epochs, 50 hidden units, $1e-3$ learning rate, 0 weight decay, no early stop) and plotted the training and validation loss (Fig. 6). We observed no overfitting, the validation loss did not increase over time, and the final test MSE was 0.161.

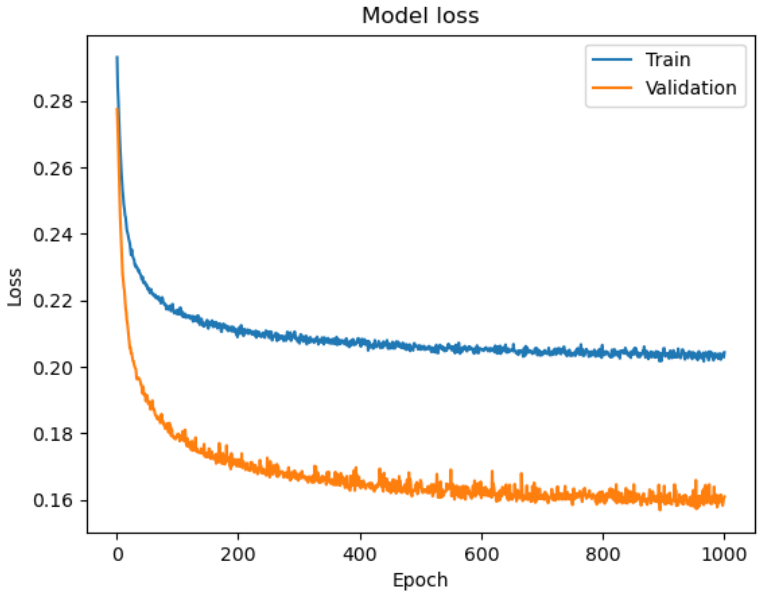

Figure 6: Fully connected neural network loss.

In an effort to improve the performance of the model, we then proceeded to hyperparameter optimization. We started by trying different learning rates while keeping all the other hyperparameters constant (50 epochs, 400 hidden units, $1e-4$ weight decay), and obtained the lowest test MSE of 0.091 with a learning rate of $2e-4$ (Fig. 7).

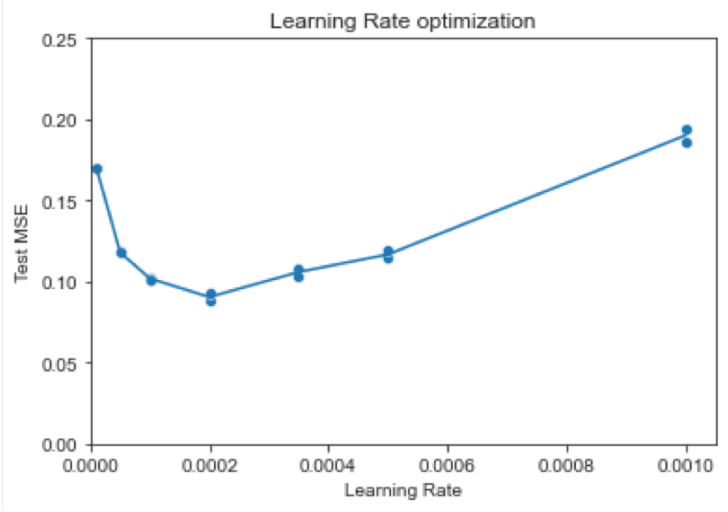

Figure 7: Learning rate optimization.

Next, we performed grid search (50 epochs, $2e-4$ learning rate) with different number of hidden units and weight decay rates. Our neural network performed best (MSE = 0.051)

with 2000 hidden units and no weight decay (Fig. 8). Weight decay rates of 0 and $1e-5$ resulted in similar test loss for all numbers of hidden units.

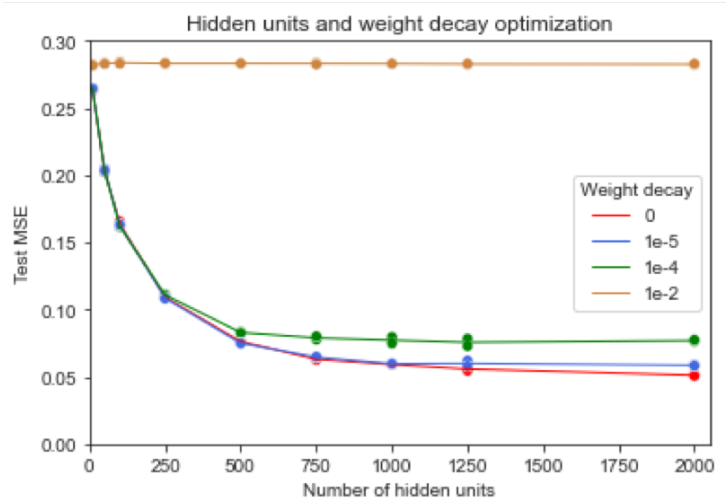

Figure 8: Hidden units and weight decay optimization.

We also looked at different window sizes during the processing of the data, for which the mean enrichment value was calculated. We tried windows of size 51, 101, 201 and 401 (50 epochs, 1000 hidden units, $1e-5$ weight decay, $2e-4$ learning rate), and the lowest lost was found with windows of size 401 with an MSE of 0.044 (Fig. 9).

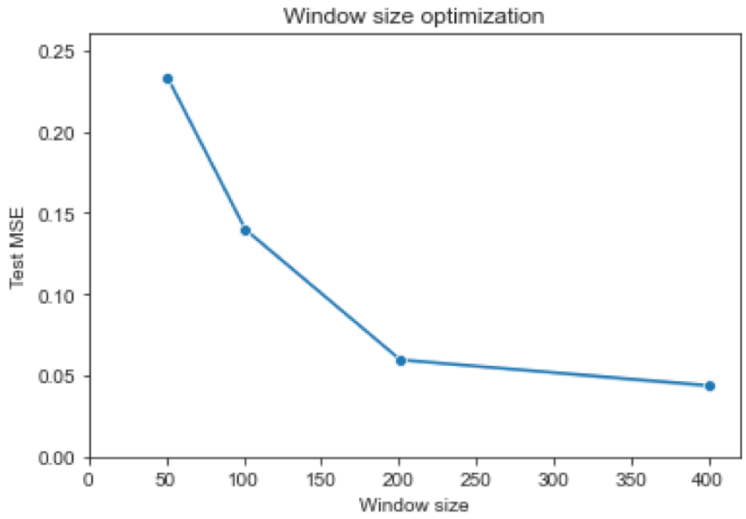

Figure 9: Window size optimization.

With these hyperparameters optimized, we built a final fully connected neural network (1000 hidden units, $2e-4$ learning rate, $1e-5$ weight decay) and let it ran for 300 epochs to see how the train and validation loss behave (Fig. 10). We can observe the validation MSE growing

and surpassing the training MSE at approximately epoch 130, indicating overfitting. With early stopping, the test loss obtained was 0.066 at epoch 133.

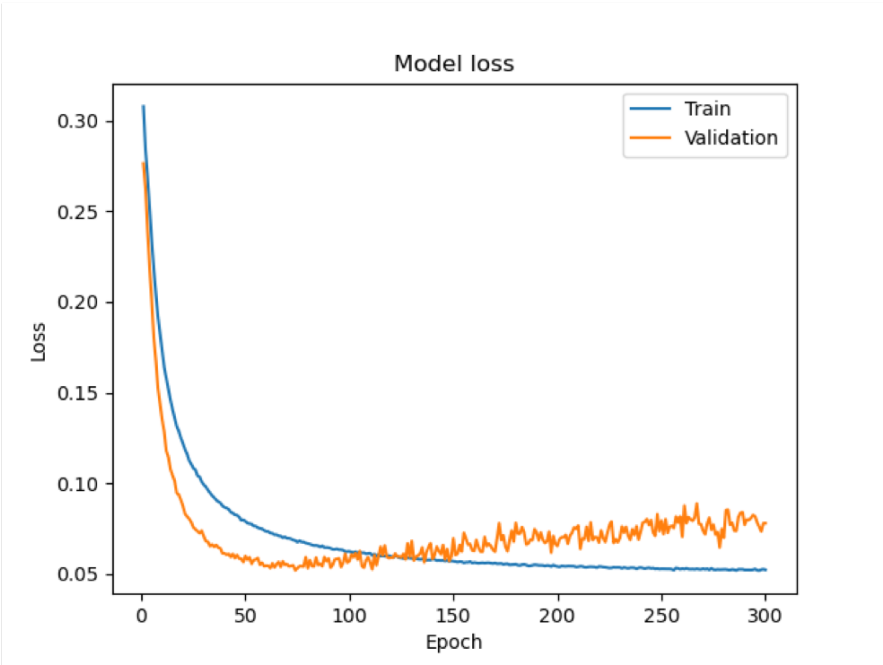

Figure 10: Optimized fully connected neural network loss.

Finally, we opted to add a second hidden layer to the neural network to check whether an extra layer brings better performance. The model was trained for 300 epochs (1000 hidden units in layer 1 and 2, $2e-4$ learning rate, $1e-5$ weight decay) (Fig. 11). The test MSE was 0.076 at epoch 141 after early stopping.

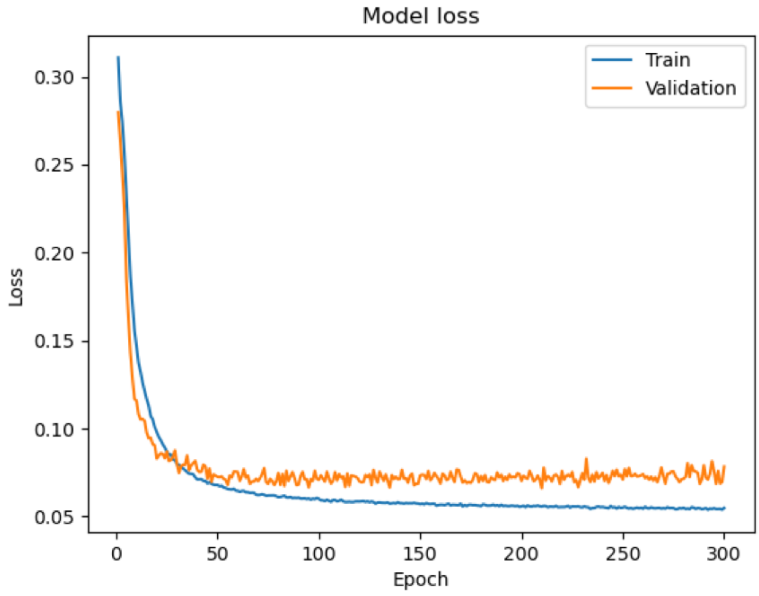

Figure 11: Optimized fully connected neural network loss with 2 hidden layers.

Convolutional neural network and hyperparameter optimization

We designed a convolutional neural network with one convolutional layer and one hidden layer using some of the hyperparameters optimized in our fully connected network, and tuned some convolutional network specific hyperparameters. As previously, we started with arbitrary values (1000 epochs, 50 convolution output channels, 100 convolution kernel size, 2 max-pooling filter size, 4 stride, 50 hidden units, $2e-4$ learning rate, $1e-5$ weight decay, no early stop), and obtained a final test MSE of 0.055 (Fig. 12).

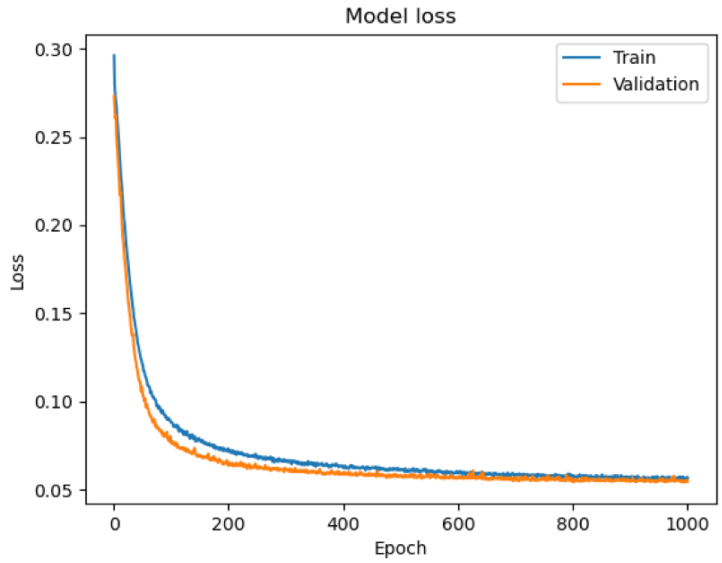

Figure 12: Convolutional neural network loss.

We started by optimizing the number of output channels in the convolutional layer (50 epochs, 100 convolution kernel size, 2 max-pooling filter size, 4 stride, 50 hidden units, $2e-4$ learning rate, $1e-5$ weight decay) and discovered that 10 channels optimized the performance of the model (MSE = 0.047) (Fig. 13).

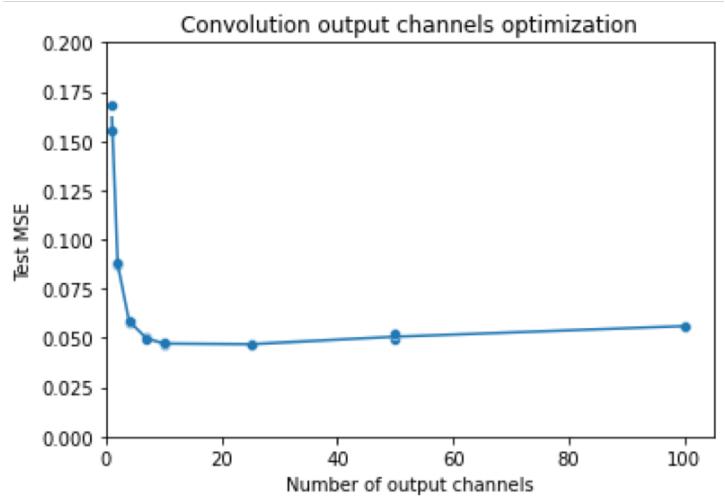

Figure 13: Convolution output channels optimization.

The size of the kernel for the convolutions was also optimized, showing a decrease in model performance as the filter size increases (50 epochs, 10 convolution output channels, 2 max-pooling filter size, 4 stride, 50 hidden units, $2e-4$ learning rate, $1e-5$ weight decay). We found the best kernel size to be 40, giving a loss of 0.047 (Fig. 14).

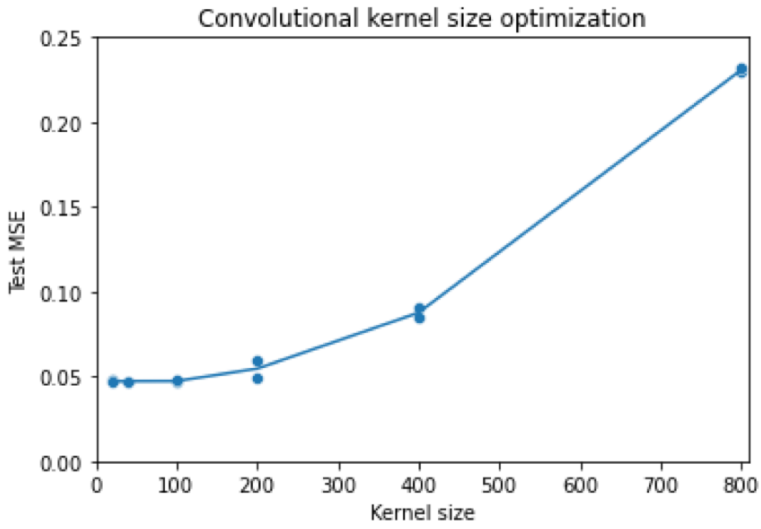

Figure 14: Convolutional kernel size optimization.

Lastly, we optimized the number of units in the hidden layer (50 epochs, 10 convolution output channels, 40 convolution kernel size, 2 max-pooling filter size, 4 stride, $2e-4$ learning rate, $1e-5$ weight decay). As in our fully connected layer, the test MSE decreased as the number of hidden units increased, and stabilized around 1000 units (MSE = 0.047) (Fig. 15).

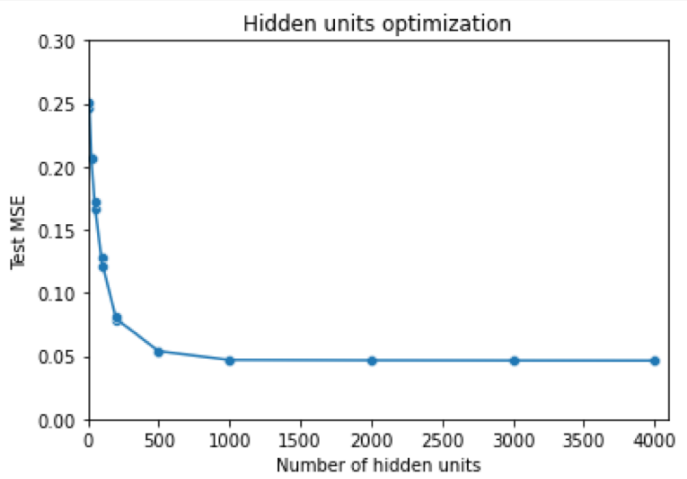

Figure 15: Hidden units optimization.

We used all the optimized hyperparameters to build a final convolutional neural network (300 epochs, 10 convolution output channels, 40 convolution kernel size, 2 max-pooling filter size, 4 stride, 1000 hidden units, $2e-4$ learning rate, $1e-5$ weight decay) and plotted the training and validation error (Fig. 16). With early stopping triggered at epoch 62, the test loss was 0.047.

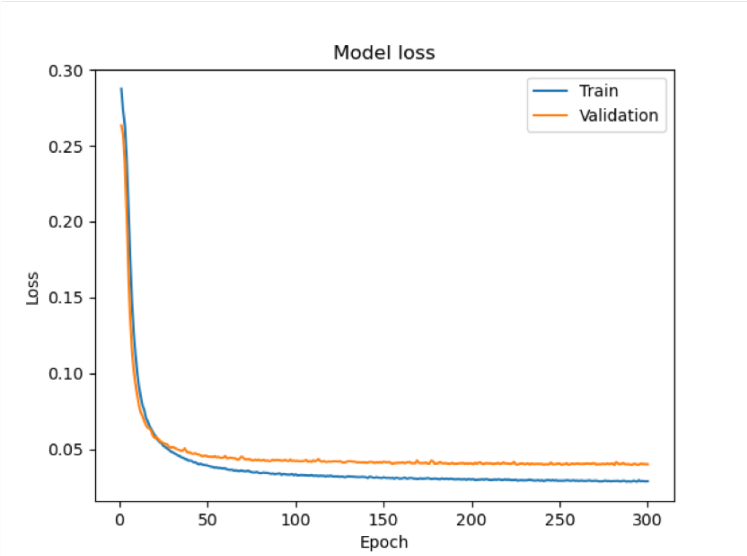

Figure 16: Optimized convolutional neural network loss.

Promoter region predictions

In order to test and validate our model for predicting chromatin accessibility, we compared signal enrichment predictions made on promoter and non-promoter regions of chromosome

22. We chose the convolutional neural network that showed best results after optimizing hyperparameters (50 epochs, 10 convolution output channels, 40 convolution kernel size, 2 max-pooling filter size, 4 stride, 1000 hidden units, $2e-4$ learning rate, $1e-5$ weight decay). A total of 1483 windows were discovered to be in a promoter region, and their signal enrichment was compared with the rest of the windows in non-promoter regions (Fig. 17). The signal enrichment average in promoter regions was 1.130 ± 0.001 (mean $\pm$ standard error of the mean), and 0.476 ± 0.031 in non-promoter regions.

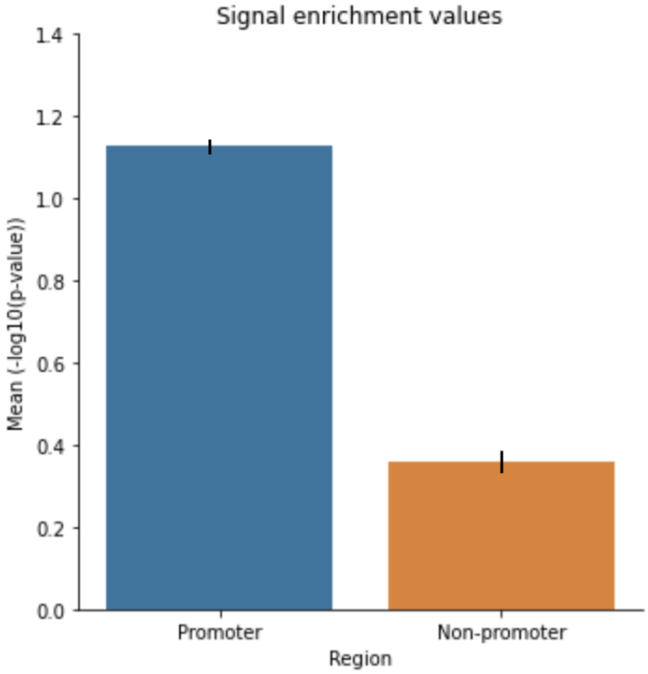

Figure 17: Signal enrichment values in promoter vs non-promoter regions of chromosome 22.

Discussion

In this project, we investigated different neural network architectures in order to find an accurate model to predict chromatin accessibility. We used the DNA sequence of chromosome 22 and adrenal gland DNase-seq measurements, and built a neural network that is able to predict signal enrichment values from the genomic sequence. With hyperparameter optimization, we successfully improved the accuracy of our models, and we tested our best performance neural network by predicting chromatin accessibility in promoter and non-promoter regions.

We first pre-processed the data by converting the DNA sequence into binary vectors using one-hot encoding, creating windows of size 201, eliminating windows with missing values or unidentified nucleotides and filtering hard-to-map regions. We transformed the data using the hyperbolic arcsine transformation and divided the data set in three partitions to train, validate and test the models.

We started by building a simple fully connected neural network with one hidden layer, and optimized certain hyperparameters that resulted in more accurate predictions. The learning rate was the first hyperparameter that we tuned, since it is believed to be one of the most important parameters in a neural network [4]. The learning rate controls the step size or amount that the weights are updated during gradient descent, moving towards a local minimum of the loss function. We discovered that a learning rate of $2e-4$ resulted in the lowest test MSE, and we kept that rate for all the models. Next, we investigated the best number of hidden units and weight decay rates together, and found the lowest loss with 2000 hidden units and no weight decay. A large number of hidden units increases the complexity of the model and the computation cost, and we observed that after 1000 hidden units, the reduction in test MSE was rather small when increasing the hidden units. Weight decay works by adding a penalty term to the loss function, which shrinks the weights during backpropagation. Our experiments showed that either no penalty or a very small penalty ($1e-5$) helped the most in reducing the error of the models. Having optimized these hyperparameters, we designed an optimized fully connected neural network (Table 3) and obtained a final test MSE of 0.066, a fair improvement over our first neural network with random hyperparameters (MSE = 0.161). We additionally implemented a fully con-

nected neural network with two hidden layers, which did not improve the predictions (MSE = 0.076) of the neural network with one hidden layer.

We moved on to optimize a convolutional neural network, and used some of the same hyperparameters that were already tuned in the fully connected neural network, such as learning rate, weight decay and number of hidden units. We first looked at the number of output channels in the convolutional layer. Each of them learns different features of the data, and also defines the number of kernels. Our tests showed that 10 is the optimal amount of convolutional output channels for our model, adding more channels did not improve the model and increased the running time. Large sizes of the convolutional filter were shown to decrease model performance, with the lowest loss found with size 40. Lastly, we looked at the number of hidden units in the hidden layer, and as in the fully connected neural network, the loss stabilized at around 1000 hidden units. The optimized convolutional neural network (Table 3) obtained predictions with a test MSE of 0.047. Hence, the convolutional neural network performed slightly better than the fully connected neural network.

Hyperparameter	Fully connected neural network	Convolutional neural network
Learning rate	$2e-4$	$2e-4$
Hidden units	2000	1000
Weight decay	$1e-5$	$1e-5$
Early stopping epoch	130	62
Test MSE	0.066	0.047
Convolutional output channels	–	10
Kernel size	–	40
Max-pool filter size	–	2
Stride	–	4

Table 3: Summary of the optimized hyperparameters.

With these parameters our neural networks perform much better than the baseline statistical models that we implemented. Predictions made using random forests outperformed by far the rest of the baseline models, and achieved results similar to neural networks. This

might indicate that linear models are not appropriate for this kind of data. On the other hand, both random forest and neural networks required a lot of computation time for training compared to the baseline models. An important disadvantage was the effort of having to manually optimize every hyperparameter. An automated approach could be developed where an algorithm tests for different parameters automatically and stores the ones with the best result. It is also worth mentioning that these hyperparameters depend on one another, and once one is modified, other hyperparameters previously fixed might become suboptimal and affect the performance of the model. An approach to fix this flaw is using grid search, which was used in this project to optimize two hyperparameters at the same time, but might become exponentially more time consuming as more parameters and values are added.

Lastly, we selected the neural network with the lowest test loss and validated the model by predicting chromatin accessibility in promoter vs non-promoter regions of chromosome 22. The predictions obtained from our model correlate with the theory. An open chromatin state is expected to be observed in promoter regions in order for them to be operative during transcription, therefore it is expected that our model should also give a response correlated to the theory. Our model predicts a signal enrichment mean of 0.3568 for non-promoter regions, and a mean of 1.1301 for promoter regions. These values report the statistical significance of enrichment (-$\log_{10}$ p-value) for each window, thus higher enrichment values meaning increased chromatin accessibility. We recognize the fact that non-promoter regions contain other genomic elements with regulatory functions, such as enhancers and silencers, which should also present higher chromatin accessibility. But we hypothesize that non-regulatory elements take up the majority of space in non-promoter regions, hence the lower chromatin accessibility profile.

The follow up steps of this project would be to test more models with more hidden layers and convolutions. Our models are relatively simple taking only one hidden layer, and on our convolutional neural network only one convolution. More layers and convolutions might be able to identify more patterns on our data and could improve the overall performance at the cost of more computational requirements. Models with no weight decay were not studied, even though having no penalty showed excellent results. We also reported better results with a window size of 401, and this could be further explored to built models with better performance potentially. With these more powerful models, motifs for known and

novel DNA binding proteins could be identified.

Our models were built using data from chromosome 22 and the adrenal gland track, hence they are most useful for making predictions on new data with the same characteristics. Additional experimentation with additional chromosomes and DNase-seq measurements from different cell lines could be explored, in order to build more specific models, or a unique flexible model able to predict over different tracks and chromosomes. Furthermore, our models could be also applied to study the effects of mutations and single nucleotide polymorphisms (SNPs) on chromatin accessibility. Mutations and SNPs can either increase or decrease the accessibility, and an in silico approach using neural networks could shed some light to this mechanism.

As a final conclusion, we can state that the neural networks we implemented, even if simple, allow us to get precise predictions on the accessibility state of the chromatin from just a DNA sequence. The hyperparameter optimization proved useful for enhancing the performance of the model, and we were able to predict a higher enrichment signal in known regulatory regions of the genome. This opens the door for further studies using neural networks as their backbone. When it comes to the results obtained, we can conclude that there is indeed useful information in the genomic sequence which can be informative of whether chromatin is in its open or closed state, and thus, it also indicates the importance of that region in gene regulation and transcription.

Bibliography

[1] David R Kelley, Jasper Snoek, and John L Rinn. "Basset: learning the regulatory code of the accessible genome with deep convolutional neural networks". In: *Genome research* 26.7 (2016), pp. 990–999.

[2] The ENCODE Project Consortium. "An integrated encyclopedia of DNA elements in the human genome". en. In: *Nature* 489.7414 (Sept. 2012), pp. 57–74.

[3] Bradley Efron and Trevor Hastie. "Chapter 18 - Neural Networks and Deep Learning". In: *Computer age statistical inference*. Cambridge University Press, 2016.

[4] Daniel Voigt Godoy. *Deep Learning with PyTorch Step-by-Step A Beginner's Guide*. 1st ed. Daniel Voigt Godoy, 2021. URL: http://leanpub.com/pytorch.

[5] *ReLU — PyTorch 1.10.1 documentation*. URL: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html (visited on 01/10/2022).

[6] *What are Convolutional Neural Networks?* en-us. URL: https://www.ibm.com/cloud/learn/convolutional-neural-networks (visited on 01/03/2022).

[7] Mehran Karimzadeh et al. "Umap and Bismap: quantifying genome and methylome mappability". In: *Nucleic Acids Research* 46.20 (Aug. 2018), e120–e120. ISSN: 0305-1048. DOI: 10.1093/nar/gky677. eprint: https://academic.oup.com/nar/article-pdf/46/20/e120/26578200/gky677.pdf. URL: https://doi.org/10.1093/nar/gky677.

[8] Ryan K. Dale, Brent S. Pedersen, and Aaron R. Quinlan. "Pybedtools: a flexible Python library for manipulating genomic datasets and annotations". In: *Bioinformatics* 27.24 (Sept. 2011), pp. 3423–3424. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btr539. eprint: https://academic.oup.com/bioinformatics/article-pdf/27/

24/3423/16901302/btr539.pdf. URL: https://doi.org/10.1093/bioinformatics/btr539.

[9] Aaron R. Quinlan and Ira M. Hall. "BEDTools: a flexible suite of utilities for comparing genomic features". In: *Bioinformatics* 26.6 (Jan. 2010), pp. 841–842. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btq033. eprint: https://academic.oup.com/bioinformatics/article-pdf/26/6/841/16897802/btq033.pdf. URL: https://doi.org/10.1093/bioinformatics/btq033.

[10] Jacob Schreiber et al. "Avocado: a multi-scale deep tensor factorization method learns a latent representation of the human epigenome". en. In: *Genome Biol.* 21.1 (Mar. 2020), p. 81.

[11] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[12] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[13] Aarhus University. *GenomeDK*. URL: https://genome.au.dk/.

[14] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.