



AARHUS  
UNIVERSITY

BIOINFORMATICS RESEARCH CENTRE (BIRC)

Project in Bioinformatics

***Implementation of Feed-Forward Neural Networks for Classification of  
Donor and Acceptor Splice Sites in Human DNA sequences***

Kristian Ozol  
201206803

Supervisor:  
Christian Storm Pedersen

*Bioinformatics Research Centre (BiRC)  
Aarhus University  
Denmark  
June 6<sup>h</sup>, 2019*

## Table of Contents

Abstract.....	3
Introduction .....	4
Theory .....	5
Neural Network Architecture and the Neuron.....	5
Activation Functions.....	7
Implementing feed forward architecture and activation functions .....	8
Loss Functions and Training by Gradient Descent using Backpropagation .....	11
Implementing Loss Functions and Optimizers.....	13
L1/L2 Regularization and Implementation .....	14
Dropout and its Implementation.....	14
Performance Metrics .....	15
Data and Experiments .....	16
Model 1 - Basic One Hidden Layer .....	19
Model 2 – Basic Five Hidden Layers .....	20
Model 3 – Five ReLU Hidden Layers .....	21
Model 4 – Five ReLU Hidden Layers with L2 Regularization.....	21
Model 5 – Five Hidden Layers with the Adam Optimizer .....	22
Model 6 – Three ReLU Hidden Layers with Dropout Regularization and the Adam Optimizer .....	23
Discussion .....	24
Results in Comparison to Recent Literature.....	24
Model Capacity .....	26
Overfitting.....	27
Vanishing Gradients .....	27
Hyperparameter Search and Performance .....	28
Conclusion.....	29
Bibliography .....	30

# Abstract

Prediction of splice sites in DNA sequences is a difficult challenge due to the manifold layers of complexity that surround the cellular exon/intron border delineation. However, machine learning approaches have shown great promise. This paper aims to demonstrate and elaborate on the implementation of various feed-forward neural networks for classification of DNA sequence windows of 140 bases as either acceptor, donor, or null splice sites. The task was split into two separate binary classification problems, and the curated Homo Sapiens Splice Sites Dataset (HS3D) was used for training. To achieve these goals, the open-source machine learning library Tensorflow and the high-level wrapper Keras were used to efficiently implement different architectures and optimize the relevant hyperparameters. Central concepts relating to the inner workings and instantiation of neural networks with these tools, such as activation, cost function, weights, and regularization, are introduced, and their relation to performance, model capacity, overfitting, and vanishing gradients was demonstrated, with the implemented models serving as examples. The evaluation of the models' performances in the task of classifying splice sites was done using 10-fold cross validation in combination with a Lock Box approach, utilizing Matthew's Correlation Coefficient as a balanced metric. Finally, the results are compared to other machine learning models presented in recent literature using the same data set, where this paper's best performing model was shown to be fairly competitive, though a more advanced neural network wins out.

# Introduction

The pre-messenger ribonucleic acid (RNA) splicing process is a fundamental feature of gene expression in eukaryotic organisms, and the result is the removal of non-coding sequences, the introns, from the initial transcript, leaving a sequential ligation of the coding sequences, the exons, to be further processed and translated into protein. In this context, the boundary between the anterior exon and the subsequent intron is called the donor splice site, while the boundary between the same intron and the following exon is called the acceptor splice site. The spliceosome is the cellular machinery in charge of conducting the splicing. At the most basic level of analysis, it identifies acceptor and donor splice sites through two short consensus sequences, corresponding to the deoxyribonucleic acid (DNA) sequences AG|GTRAGT and (Y)nNCAG|G (the | sign indicates the cut site) for the donor and acceptor, respectively, in IUPAC notation [1]. However, an enormously complex interplay of factors like epigenetic modifications, silencer/enhancer sequences, and the general genomic context also influences whether or not a given potential site will be used in the splicing process. As such, sequence elements up to 50 nucleotides away from the boundary, and possibly even further, can influence the spliceosome's decision [2].

The delineation of exon/intron boundaries is further complicated by the phenomenon of alternative splicing, which leads to a large variety of messenger RNAs from the same gene due to variation in which donor and acceptor sites are chosen in the splicing procedure. This can be influenced not just by cis-acting sequence elements but also by various trans-acting factors such as cellular signaling pathways, levels of various splicing proteins, and the cell type in question [3].

Since splicing directly relates to how genes are subdivided into introns and exon, and in what manner these are expressed in multitude forms, it has been of utmost interest to develop computational models for the prediction of donor and acceptor sites, which has traditionally been a significant challenge owing to the now apparent complexity of how splice signals are identified in the transcription process.

Despite these issues, significant success has been achieved using a wide range of machine learning approaches. Among them, Bayesian networks [4], Support Vector Machines [5], hidden Markov models [6], and random forests [7] have all been used to predict splice sites using human DNA

sequences. However, in the last eight years or so, there has been an explosive growth in the field of artificial neural networks (NNs), with state-of-the-art models demonstrating superior performance in numerous fields of study, which is also true in the domain of bioinformatics [8]. Specifically, in relation to splice site prediction, there are a number of current papers that showcase the potential of using artificial NNs as models by attaining outstanding accuracy in their predictions [9-11], while the advanced techniques and architectures utilized in those papers also demonstrate how much the field of splice site prediction using NNs has evolved in just two decades when compared to earlier literature [12, 13].

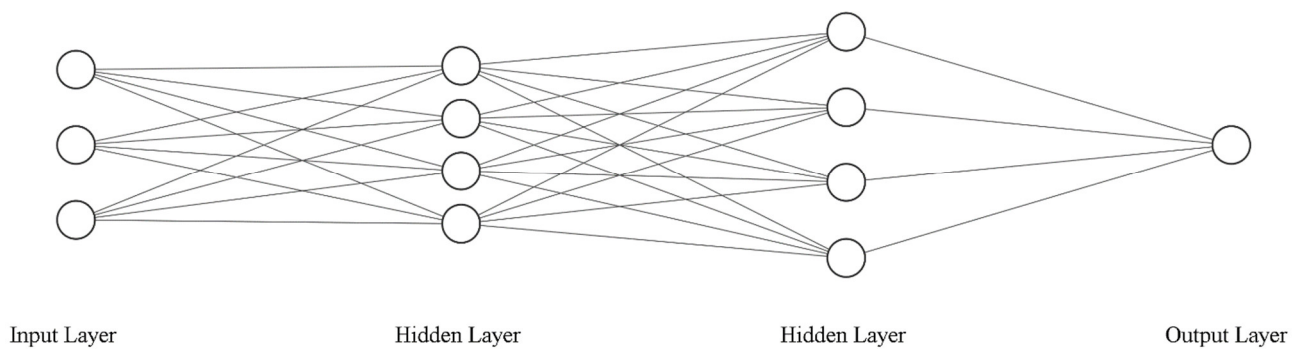
In light of how well-suited neural networks with their high model capacity seem to be for the exceedingly complex task of predicting splice sites from sequence data, this paper aims to implement a range of feed-forward artificial NN with relatively simple architecture and compare their performance to the models in the current literature, illustrating both the ease of the implementation using the Keras wrapper with Tensorflow and the relatively high effectiveness of even simple NNs. The classification problem will be subdivided into two binary ones, such that two equivalent models for the identification of each separate splice site type will be implemented.

For this purpose, DNA sequences from the Homo Sapiens Splice Sites Dataset [14] will be used with the lock box [15] approach to both train and evaluate the ANN models, using the Matthew's Correlation Coefficient [16] as the performance metric due to it being a more reliable and balanced estimate in comparison to commonly used metrics like accuracy and F1 scores [17].

## Theory

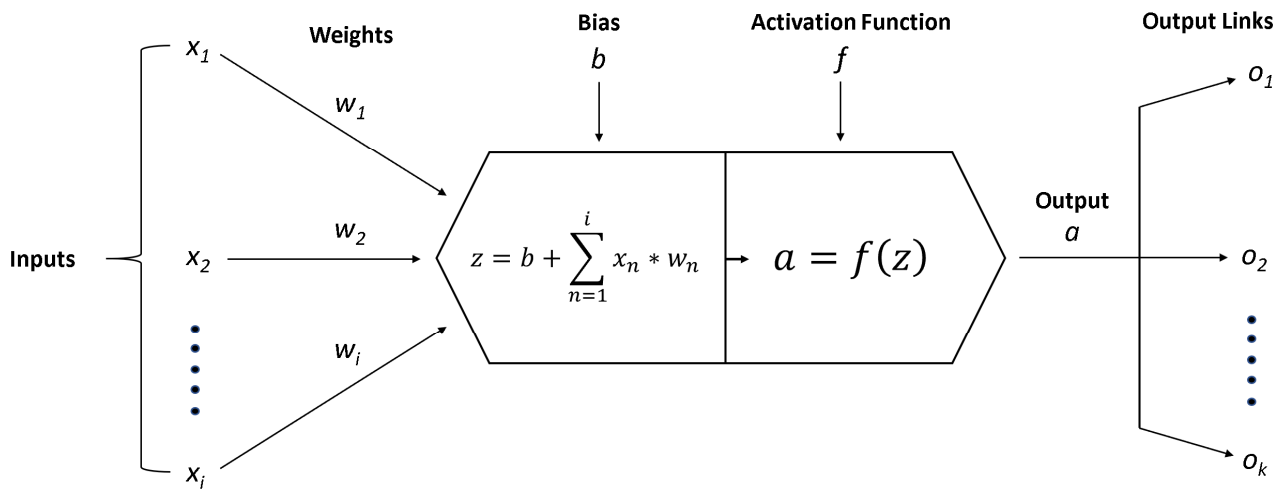
### Neural Network Architecture and the Neuron

Conceptually, artificial neural networks are made up of three parts: An input layer, an output layer, and any number of hidden layers. The wide range of possible architectures and differences in how the layers are connected give rise to the multitude of neural network categories. The focus in this paper will mainly be on comparatively simple feed-forward networks. In these networks, every node in one layer is connected to all nodes in the next, as illustrated in **figure 1**.



**Figure 1:** A feed forward artificial neural network for a regression problem where every node in layer  $k$  is connected to all nodes in layer  $k+1$ .

The most fundamental constituent of any artificial neural network is the neuron, which in relation to neural network architecture is also called a node. In essence, each neuronal unit in the hidden layers of a given network will take a number of weighted inputs, process them, and pass on the result as an output. The actual processing step can take on a myriad of forms, but the most common elements are simple numerical addition of a bias and the all-important application of a so-called activation function. **Figure 2** illustrates this with a simple schematic. Here, the outputs from the nodes in the previous layer, either input layer nodes or other hidden layer ones, are named  $x_1, x_2, \dots, x_i$ , with their respective weights being  $w_1, w_2, \dots, w_i$ . The neural processing consists of summing the weighted inputs with a bias,  $b$ , added, resulting in a sum,  $z$ , also known as the pre-activation sum. An activation function,  $f$ , is then applied to  $z$  to produce the output,  $a$ , the activation, which is passed to the nodes  $o_1, o_2, \dots, o_k$  in the next layer.



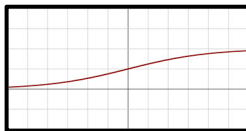
**Figure 2:** A schematic representation of a neuron in a hidden layer of a feed forward neural network, processing a number of weighted inputs by adding a bias and applying an activation function to the resulting value, thus producing the output, which is passed on to a number of nodes in the next layer.

## Activation Functions

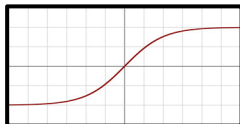
Not surprisingly, the choice of activation function does in large part dictate how a neuron will operate, which in turn influences the whole network. One important note is that much of the non-linearity inherent to neural network models stems from a non-linear activation function.

Traditionally, the Sigmoid and Hyperbolic Tangent functions have been widely used as activation functions.

Sigmoid:  $f(z) = \sigma(z) = \frac{1}{1+e^{-z}}$



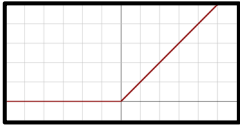
Hyperbolic Tangent:  $f(z) = \tanh(z) = \frac{(e^z - e^{-z})}{(e^z + e^{-z})}$

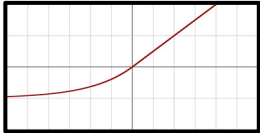


However, both functions have been shown to lead to excessive node saturation unless specific weight initialization regimes are employed, seemingly due to the way they both squash the input into a small range, (0,1) and (-1,1) for the sigmoid and hyperbolic tangent, respectively [18]. Node saturation relates to the training of the network through backpropagation, which will be explored later, but suffice to say that the model will potentially suffer from what is called vanishing

gradients, in short, an inability in the earlier layers of the network to learn from the training data. As such, it is generally discouraged to use the aforementioned functions in feed forward neural networks [19].

As prominent alternatives, the rectified linear activation function (ReLU) [20], along with even more recent additions such as the exponential linear unit function (ELU) [21], including their variations, have been used to great effect in combatting inefficient model training stemming from vanishing gradients [22] and increasing generalization performance.

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


$$\text{ELU: } f(z) = \begin{cases} \alpha(e^z - 1) & \text{for } z < 0 \\ z & \text{for } z \geq 0 \end{cases}, \text{ where } \alpha > 0$$


Furthermore, for the output layer of the network the activation function will usually be either the identity function or the softmax function, depending on whether the model is made for a regression or classification problem, respectively. The latter operates on the whole output layer by applying the exponential function on the sum,  $s$ , from each each neuron and dividing by the sum of those exponentials, thus creating a probability distribution consisting of a number of classes equal to the amount of output layer neurons, where the sum of the probabilities is equal to 1. This means that it is necessary to use two output nodes when doing binary classification using softmax, since having only a single node will lead to the output always being 1.

While some activation functions do compare favorably to others in general, their behavior might deviate from common patterns in a specific dataset or for certain problems. As such, it is prudent to experiment with several different functions when optimizing the hyperparameters of a neural network.

### Implementing feed forward architecture and activation functions

All components of a neural network described so far can easily be implemented with the Keras wrapper using Tensorflow in Python. For the following code to function, the Tensorflow package and a number of supporting ones need to be installed in the active Python environment. See the appendix for a comprehensive list of installed libraries.



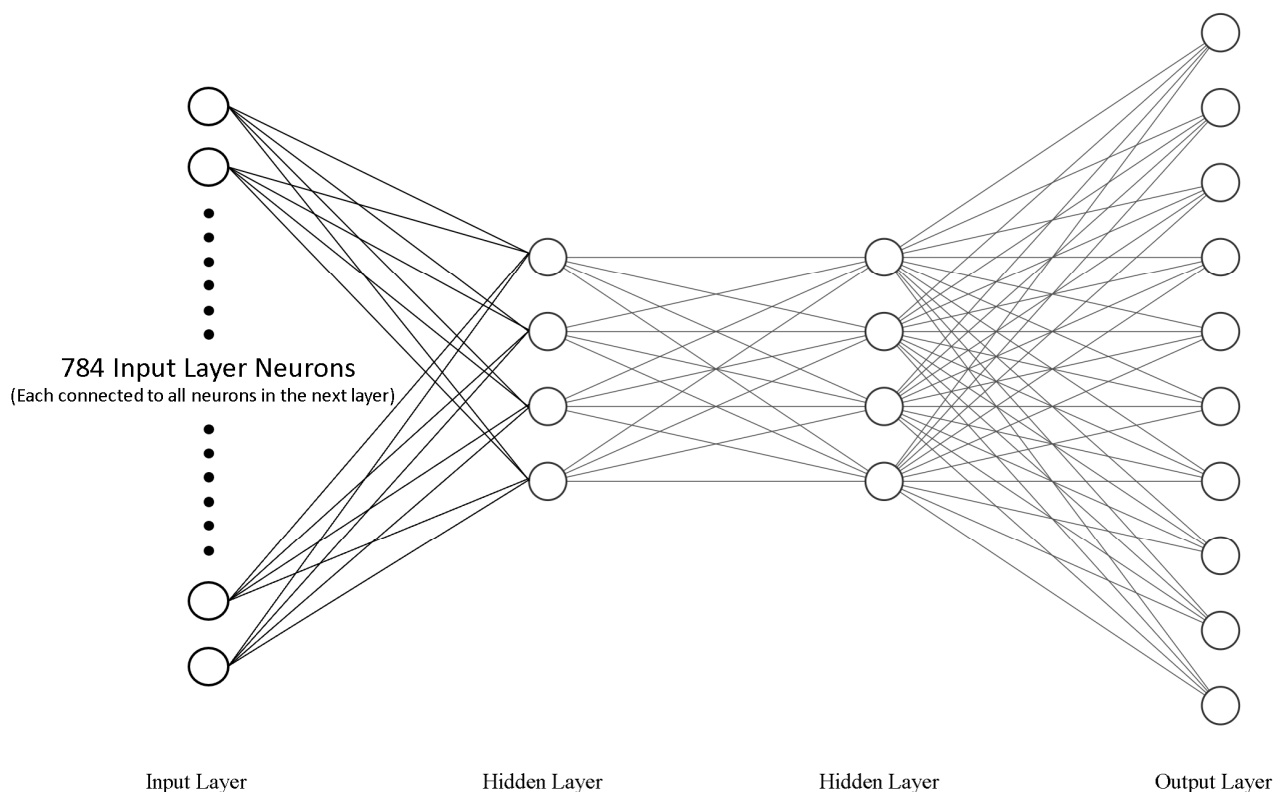
For the purposes of illustration this paper will be using the neural network dataset-equivalent of “Hello world!”: MNIST. It consists of 28x28 pixel pictures of hand-written numbers, ranging from 0 to 9. To build a simple feed forward neural network that can be trained to determine what number is depicted, the first step is to import the various Keras components that will be needed:

```
1. from tensorflow.keras.datasets import mnist
2. from tensorflow.keras.models import Sequential
3. from tensorflow.keras.layers import Dense, Activation
4. from tensorflow.keras.utils import to_categorical
```

After that, the MNIST dataset can be loaded. To fit easily into the model to come, the 28x28 pixel arrays are formatted into vectors with size 784, and the categorical labels are one-hot encoded for compatibility.

```
1. (x_train, y_train), (x_test, y_test) = mnist.load_data()
2. one_hot_labels_train = to_categorical(y_train, num_classes=10)
3. one_hot_labels_test = to_categorical(y_test, num_classes=10)
4. x_train = x_train.reshape(60000, 28*28)
5. x_test = x_test.reshape(10000, 28*28)
```

A feed forward neural network with two hidden layers, an input layer with 784 neurons, and an output layer that produces a probability for each number is depicted in **Figure 3** below.



**Figure 3:** A simple feed forward artificial neural network taking the MNIST data as input and outputting probabilities for which number was depicted.

To produce this model architecture with Keras, using the sigmoid activation function for the hidden layers and softmax for the output layer and generating neurons with bias, one simply initiates the feed forward model with `Sequential()` and add the layers, specifying the input layer dimensions in the first `Dense` addition.

```
1. model = Sequential()
2. model.add(Dense(4, input_shape = (784,), activation = "sigmoid"))
3. model.add(Dense(4, activation = "sigmoid"))
4. model.add(Dense(10, activation = "softmax"))
```

The activation parameter can also take other arguments such as “elu”, “relu”, “tanh”, and “linear”, to employ other activation functions like the ones discussed. Constructing additional layers and controlling the width of them is merely a matter of inserting more `model.add` calls and adjusting the first argument of the `Dense` function.

## Loss Functions and Training by Gradient Descent using Backpropagation

With the fundamental architectural elements of the neural network in place, what remains is to modify the weight and bias parameters using labeled training data. For this task, it is necessary to employ a function that quantifies how well the predictions,  $\hat{\mathbf{y}}$ , of the neural network approximates the response values,  $\mathbf{y}$ , in the data. This is known as a loss function, or cost function, denoted here by the notation  $C(\mathbf{w}, \mathbf{b})$ , where  $\mathbf{w}$  and  $\mathbf{b}$  stands for the weights and the biases, respectively. In binary classification problems the most commonly used loss function is called binary Cross-Entropy Loss. For a data point with any given number of predictor variables and two classes as response,  $y_1$  and  $y_2$ , the loss function can be written as follows:

$$C(\mathbf{w}, \mathbf{b}) = -(y_1 * \log(\hat{y}_1) + y_2 * \log(\hat{y}_2))$$

Which is merely a specific case of the more general multi-class Cross-Entropy loss, used in classification problems with  $n$  classes, each having a response variable  $y_i$ :

$$C(\mathbf{w}, \mathbf{b}) = - \sum_{i=1}^n y_i * \log(\hat{y}_i)$$

As for regression models, the ubiquitous mean squared error function is often used. There are for both regression and classification multiple options in the choice of loss function, but for the sake of brevity and in an attempt to limit the hyperparameter space, only the Cross-Entropy variants will be used for the rest of this report. The combination of the softmax activation function on the output layer and the use of a Cross Entropy loss function is sometimes referred to as Softmax loss.

Training the neural network is a matter of minimizing the chosen loss function, which qualifies as an optimization problem. To solve it is no easy feat however, since it is non-convex and staggeringly non-linear, with the number of variables often reaching overwhelming numbers, thus rendering it unfeasible to determine the global minimum, or even local ones, using regular analytic differentiation from calculus, greedy search, genetic algorithms, brute-force search, or other such methods [23, 24]. A conceptually simple but powerful solution to this problem is to perform gradient descent using what is called the backpropagation algorithm [25]. To quickly unpack the essential nature of these two concepts, suppose that the loss function  $C$  for any given data point

depends on  $m$  parameters,  $v_1, v_2, \dots, v_m$ , (weights and biases) and that the gradient,  $\nabla C$ , is defined as the column vector of partial derivatives, as so:

$$\nabla C = \left( \frac{\partial C}{\partial v_1}, \dots, \frac{\partial C}{\partial v_m} \right)^T$$

Given a minute change in the variables,  $\Delta v$ , represented by a column vector of changes,  $\Delta v = (\Delta v_1, \dots, \Delta v_m)^T$ , then the corresponding change in the loss function,  $\Delta C$ , can be approximated by the following formula:

$$\Delta C \approx \nabla C * \Delta v$$

This implies that if one knows the gradient of the loss function then choosing  $\Delta v$  to be  $-\eta \nabla C$ , where  $\eta$  is called the learning rate, will result in a decrease in  $\Delta C$ , as shown in the following formula:

$$\Delta C \approx \nabla C * \Delta v \approx \nabla C * -\eta \nabla C \approx -\eta * ||\nabla C||^2$$

where the last term,  $||\nabla C||^2$ , is positive or equal zero, which means  $\Delta C$  will always either decrease or not change. Using this line of logic, gradient descent boils down to taking small incremental steps in the opposite direction of the gradient, eventually reaching a local minimum for the loss function. The way the rudimentary gradient descent method changes the parameters of the neural network, from  $v$  to  $v^*$ , can thus be encapsulated in the following update formula:

$$v^* = v - \eta \nabla C$$

To train the neural network using all the data in a given data set, the overall gradient is calculated as the mean of each data point's gradient. With  $d$  data points, each input designated  $x$ , the overall gradient is:

$$\nabla C = \frac{1}{d} \sum_x^d \nabla C_x$$

However, using every single data point for each update step is computationally expensive, as it requires both a so-called forward pass through the network to compute a prediction, and a

backwards pass to calculate the gradient, for each observation, which can easily number in the tens of thousands, or even more. This is a problem which is largely solved by the application of the widely used stochastic gradient descent (SGD) optimizer. The idea is simply to use a small subset, in this context called a mini-batch, of the data to estimate the gradient in each iteration of updating the parameters of the network. Each subsequent update uses a new mini-batch, and when the training process in this way has used all observations in the data it will have completed what in NN terminology is called an epoch. The number of epochs that the model will train for is a hyperparameter in and of itself. The SGD optimizer will act as a baseline for this project, but it leaves a lot to be desired in terms of training efficiency. To improve upon the basic concept of SGD, an assortment of optimizer variants have been developed, but among them the most ubiquitous and broadly successful is the adaptive moment estimation (Adam) optimizer [26], the core feature of which is that it adds momentum to the update cycle, while also tuning the individual learning rates of the models' parameters [27]. To further develop the Adam algorithm, a recent paper incorporated a more advanced form of momentum called Nesterov Accelerated Gradient into the update step, calling it Nesterov-accelerated Adaptive Moment Estimation (Nadam) [28]. Several other optimizers are available as well, but SGD and Adam will be the focus in this project, due in large part to their prominence.

As for the actual calculation of the gradient, that's where backpropagation comes into play. The wonderful mathematical intricacies of the algorithm are beyond the scope of this report, but in short it achieves high computational efficiency by first differentiating the loss function and then working backwards through the network, applying the differential calculus chain rule to each layer and using the partial derivative to calculate the next. In this way, the error found by the loss function is propagated backwards throughout the network and used to determine the gradient, hence the name.

### **Implementing Loss Functions and Optimizers**

As a testament to the high abstraction level of the Keras wrapper, the implementation of both the loss function and the optimizer can be done with only one line of code using the `model.compile` function, which can also be further customized with a plethora of additional arguments.

```
1. model.compile(optimizer='SGD', loss='categorical_crossentropy')
```

Loss functions like Kulback Leibler Divergence and Hinge can be used by replacing 'categorical\_crossentropy' with 'kullback\_leibler\_divergence' and 'hinge', respectively. Using other optimizers such as Adam and Nadam is done simply by replacing 'SGD' with 'adam' or 'nadam'.

### L1/L2 Regularization and Implementation

Overfitting is one of the greatest threats to the generalization ability of a feed forward neural network due to its impressive model capacity. If one does not tread carefully, such a network will easily capture and adapt to the noise in a given data set and lose sight of the underlying patterns [29]. To combat this issue, L1 and especially L2 regularization are some of the most common tools, the latter often being called weight decay in the context of neural networks. Both types apply a penalty to the cost function based on the size of the weights in the networks, like so:

$$L1: C(w, b) = C_0 + \frac{\lambda}{n} * \sum_w ||w|| \quad L2: C = C_0 + \frac{\lambda}{2n} * \sum_w ||w||^2$$

With  $\lambda$  being penalty tuning parameter and  $C_0$  being the unmodified cost. Intuitively, these penalties help ensure that the weights in the network only grow to the extent that it sufficiently decreases the loss value, resulting in improved generalization [30].

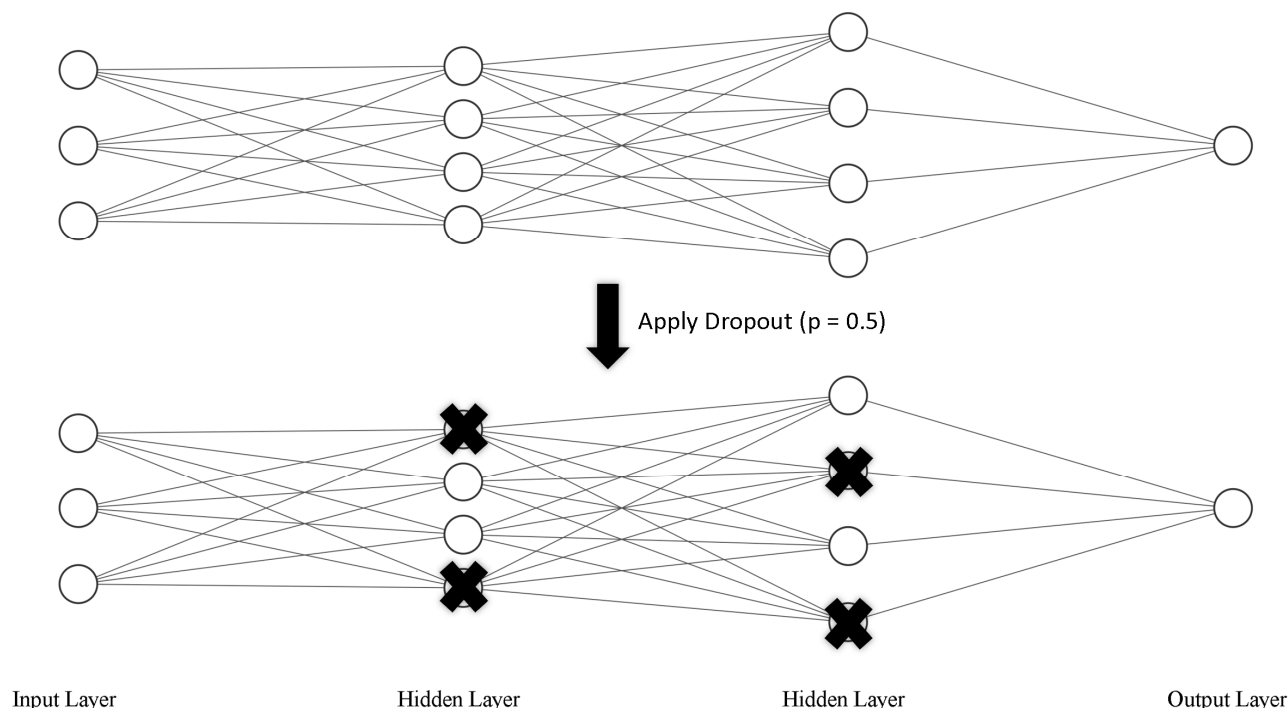
With Keras, adding L1/L2 regularization is done by passing either `regularizers.l2( $\lambda$ )` or `regularizers.l1( $\lambda$ )` as the `kernel_regularizer` argument in each layer's `model.add()`.

```
1. model = Sequential()
2. model.add(Dense(4, input_shape = (784,), activation = "sigmoid", kernel_regularizer= regularizers.l2(0.01)))
3. model.add(Dense(4, activation = "sigmoid", kernel_regularizer= regularizers.l2(0.01)))
4. model.add(Dense(10, activation = "softmax"))
```

### Dropout and its Implementation

What has arguably become the most successful form of regularization in the domain of neural networks is the very simple concept called dropout [31]. This usually involves that each node in the hidden layers of the network has a certain probability,  $p$ , of being temporarily removed from each mini-batch used in the stochastic gradient descent, including all its incoming and outgoing connections. Effectively, this is akin to model averaging by training many different networks and employing consensus-based prediction, which, not surprisingly, tends to reduce overfitting. When using the trained model, in order to compensate for the fact that all nodes are now present, all

weights are multiplied by  $p$ , the chosen probability of dropout. **Figure 4** illustrates the application of dropout on a simple feed forward neural network for a single mini-batch:



**Figure 4:** A simple feed forward artificial neural network with dropout applied, using a dropout rate,  $p$ , of 0.5.

It is also possible to apply dropout to the input layer, though the dropout rate in that case should remain very low [31].

Adding dropout with dropout rate  $p$  to the hidden layers of a neural network with the Keras wrapper can be done by using `model.add(Dropout(p))` between each hidden layer and the output layer:

```
1. model = Sequential()
2. model.add(Dense(4, input_shape = (784,), activation = "sigmoid", kernel_regularizer= regularizers.l2(0.01)))
3. model.add(Dropout(0.5))
4. model.add(Dense(4, activation = "sigmoid", kernel_regularizer= regularizers.l2(0.01)))
5. model.add(Dropout(0.5))
6. model.add(Dense(10, activation = "softmax"))
```

## Performance Metrics

As hinted at in the introduction, the primary metric that will be used in this project is Matthew's Correlation Coefficient (MCC). It is a performance score for binary classification problems that is

highly balanced, in that it considers all classes of the confusion matrix, and as such will only approach high values when both the positive and negative predictions tend to be correct and in the same proportions. This is mostly relevant with unbalanced data sets where one of the labels is being overrepresented, but it is good practice to use MCC in any case. The numerical range of MCC is from -1, the worst, to +1, the perfect performance. The formula is as follows [17]:

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) + (TN + FN)}}$$

Where TP, TN, FP, FN are the counts of true positive, true negative, false positive, and false negative. Since MCC is not part of the standard performance metrics in Keras, the `matthews_corrcoef` function was imported from the SciKit Learn library and used in conjunction with the `predict` function of Keras to produce the MCC score after the completion of the model training. Due to the MNIST example being a multiclass classification problem, MCC isn't directly applicable in that setting.

## Data and Experiments

The HS<sup>3</sup>D is a procured dataset consisting of Homo Sapiens DNA sequences extracted from GenBank (Rel. 123), each entry having a length of 140 nucleotides and representing either a true or false splice site. The samples are further subdivided into donor (Exon-Intron, EI for short) or acceptor (Intron-Exon, IE for short) categories. Both true and false EI sites have the canonical GT nucleotides at position 71 after the potential splice site, and correspondingly in all IE sequences the canonical AG is found at position 69, just before the potential splice site. An example of each is shown in **Figure 5**:



<b>EI True splice site</b>	
Position:	... 66 67 68 69 70 71 72 73 74 75 76 77 ...
Nucleotide:	... T C A G T <span style="border: 1px solid black;">G T</span> A A G T A ...
<hr/>	
<b>EI False splice site</b>	
Position:	... 66 67 68 69 70 71 72 73 74 75 76 77 ...
Nucleotide:	... T A G A C <span style="border: 1px solid black;">G T</span> G C G G C ...
<hr/>	
<b>EI True splice site</b>	
Position:	... 64 65 66 67 68 69 70 71 72 73 74 75 ...
Nucleotide:	... C C T G C <span style="border: 1px solid black;">A G</span> G A G G C ...
<hr/>	
<b>EI False splice site</b>	
Position:	... 64 65 66 67 68 69 70 71 72 73 74 75 ...
Nucleotide:	... C C T A C <span style="border: 1px solid black;">A G</span> G C C C A ...

**Figure 5:** Examples of each of the subcategories in the HS<sup>3</sup>D dataset. The three dots indicate that the full sequence extends in both directions, reaching a full 140 nt length in the actual sample. The boxed nucleotides correspond to the canonical donor GT and acceptor AG motif after and before the potential splice site, respectively.

As for the number of samples, the positive EI category has 2796 sequences, and the positive IE category 2880. In contrast, the negative categories each have over 200,000 sequences. Training a network with that much data is computationally intensive, and, more importantly, the extreme class imbalance can negatively influence the generalization performance of the model. As such, only a subset of each negative category is used, with the size equaling the corresponding positive category. The table below shows the distribution of samples.

	True Donor	False Donor	True Acceptor	False Acceptor
Samples Available	2,796	271,937	2,880	332,296
Samples Used	2,796	2,796	2,880	2,880

To facilitate using the sequences as input for the training of the neural network models, one-hot encoding is employed, with each nucleotide letter being represented by a 1 for the sake of clarity. Thus, each base letter is encoded by a column vector of length 4, as illustrated in **Figure 6**:

## One-Hot Encoding Scheme

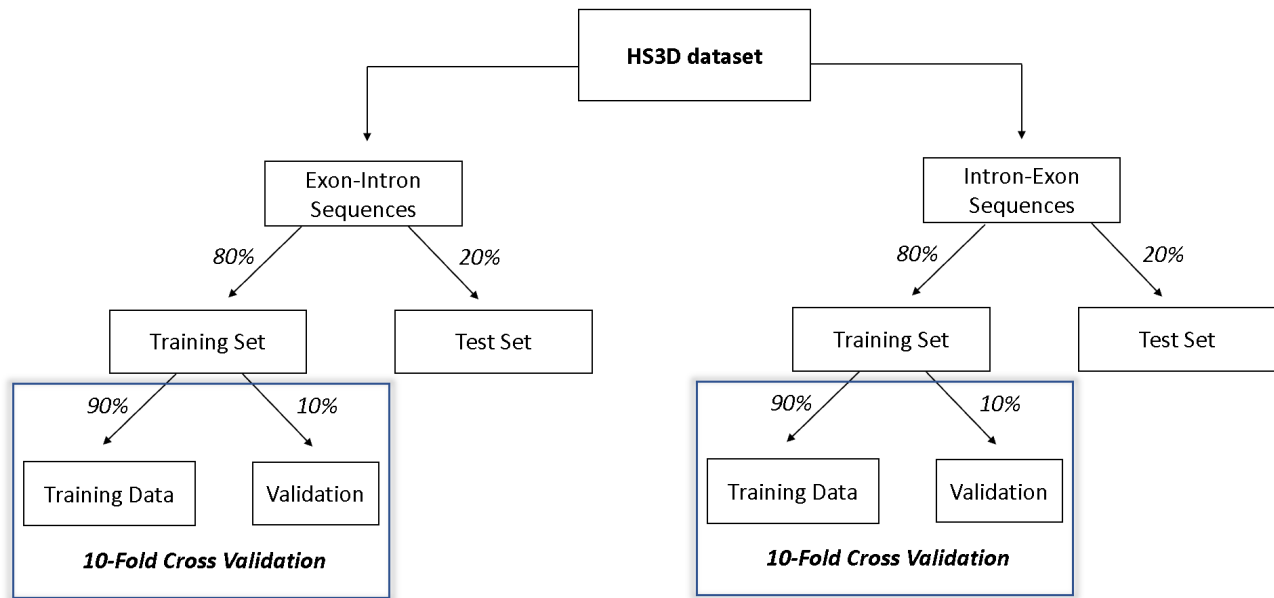
$$A = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad T = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

## Encoding of Sample Subsequence

Nucleotides:	...	T	C	A	G	T	G	T	A	A	...
One-Hot Vectors:	...	$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	...

**Figure 6:** The One-Hot Encoding scheme shows how each nucleotide letter is converted into its corresponding one-hot vector. The Encoding of Sample Sequence demonstrates how a nucleotide sequence will be encoded. The three dots indicate that the full sequence extends in both directions, reaching a full 140 nt length in the actual sample.

As stated in the introduction, the lock box approach will be employed, which entails splitting the data up into two primary partitions, the training set and the test set. The training set will be used to conduct 10-fold cross validation of every model after a varying number of rounds of hyperparameter optimization, where after the test set will be used to do an evaluation of the performance without any subsequent reconfiguration of the given model, which allows for a proper estimation of the model's generalizability as the final evaluation uses what is effectively completely new data. As per commonly suggested ratios, the test set will consist of 20% of the total samples, randomly selected but stratified by label. **Figure 7** gives an overview of the data partitions and subcategories.

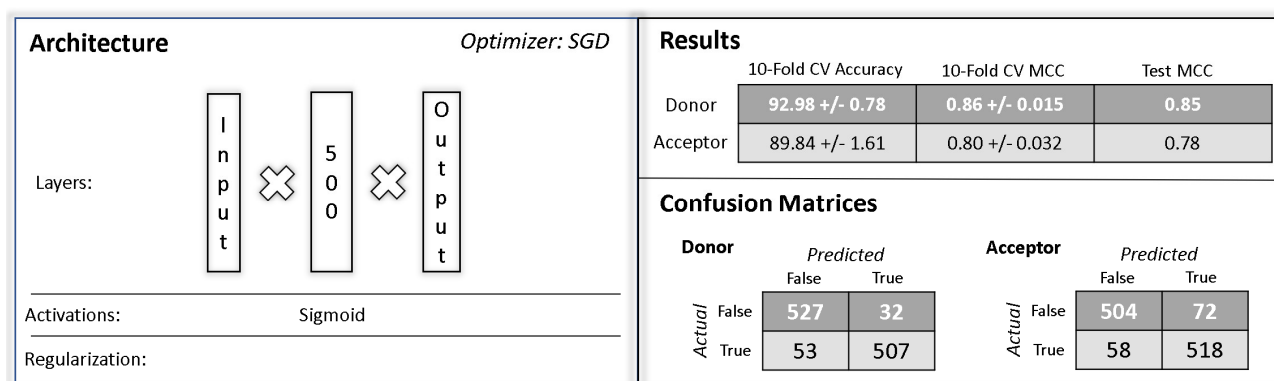


**Figure 7:** The HS<sup>3</sup>D dataset is split into its EI and IE subcategories, each containing both positive and negative samples. These subcategories are then subjected to the lock box data portioning, using 20% for a test set that will be used in the final model evaluation, and 80% to a training set that will be used for 10-fold cross validation, with each iteration using 10% of the samples for validation and 90% for training the model.

All the models in the following sections were trained for 40 epochs each. This number is somewhat arbitrary and would in most settings be highly flexible but having it fixed is for the purpose of demonstrating how various changes to a neural network affect the prediction results. See the Appendix for the full code employed.

### Model 1 - Basic One Hidden Layer

**Figure 8** below describes the configuration and results of a very basic feed-forward neural network, simply consisting of the input layer, a hidden layer, and the output layer.



**Figure 8:** Hyperparameters and results for model 1. The Architecture box illustrates the model's hyperparameters. In the layers section, the crosses represent the connections between feed-forward fully-connected layers, and the number in the box of the hidden layer indicates the number of nodes. In the Results box the CV and test results for both Exon-Intron (Donor) and Intron-Exon (Acceptor) splice site predictions are shown. Each number in the 10-Fold estimates that follows a +/- represents the standard deviation. The accuracy is given in percent. In the Confusion Matrices box both the donor and acceptor confusion matrix associated with the Test MCC score are shown.

The model was defined with Keras using the code snippet below:

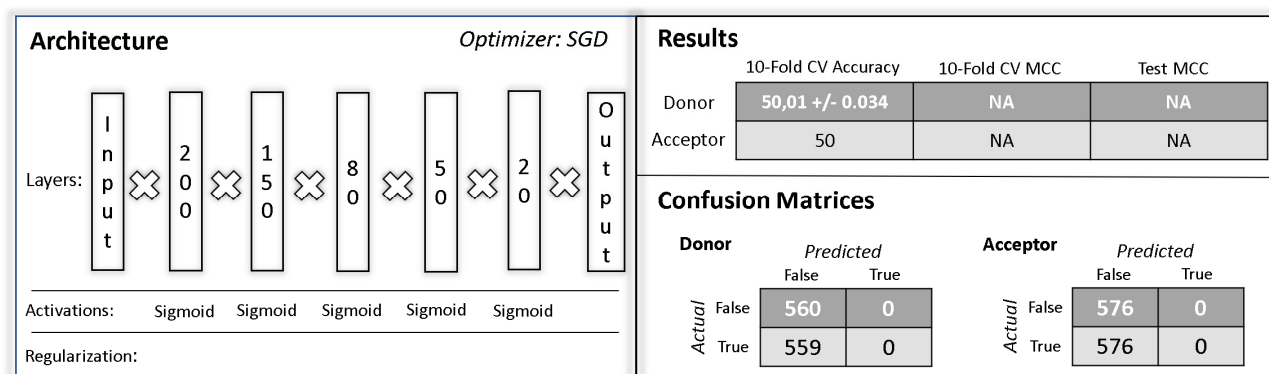
```

1. def buildModelAndCompile():
2.     K.clear_session()
3.     model = Sequential()
4.     model.add(Flatten(input_shape=(140, 4)))
5.     model.add(Dense(500, activation = 'sigmoid'))
6.     model.add(Dense(2, activation = 'softmax'))
7.     model.compile(optimizer = 'sgd', loss = 'binary_crossentropy', metrics=['accuracy'])
8.     return(model)

```

## Model 2 – Basic Five Hidden Layers

Figure 9 shows the configuration and results for a model with 5 hidden layers, but with the same total number of nodes as in model 1.



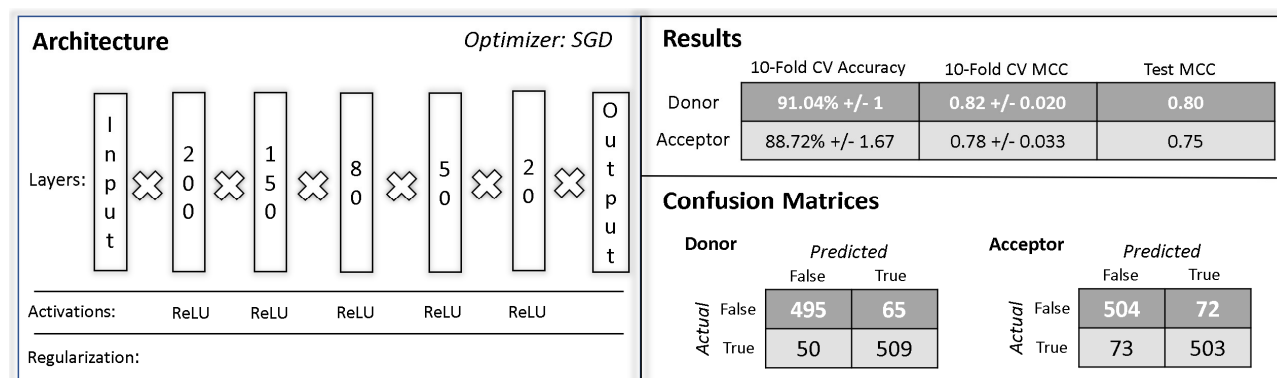
**Figure 9:** Hyperparameters and results for model 2. See figure 8 for common elements. In the Results box, NA indicates that the number couldn't be calculated, which in the case of MCC is due to one class of predictions being entirely absent.

The model was defined with Keras using the code snippet below:

```
1. def buildModelAndCompile():
2.     K.clear_session()
3.     model = Sequential()
4.     model.add(Flatten(input_shape=(140, 4)))
5.     model.add(Dense(200, activation = 'sigmoid'))
6.     model.add(Dense(150, activation = 'sigmoid'))
7.     model.add(Dense(80, activation = 'sigmoid'))
8.     model.add(Dense(50, activation = 'sigmoid'))
9.     model.add(Dense(20, activation = 'sigmoid'))
10.    model.add(Dense(2, activation = 'softmax'))
11.    model.compile(optimizer = 'sgd', loss = 'binary_crossentropy', metrics=['accuracy'])
12.    return(model)
```

### Model 3 – Five ReLU Hidden Layers

Figure 10 shows the configuration and results for a model exactly like model 2, but with ReLU activations on each hidden layer.



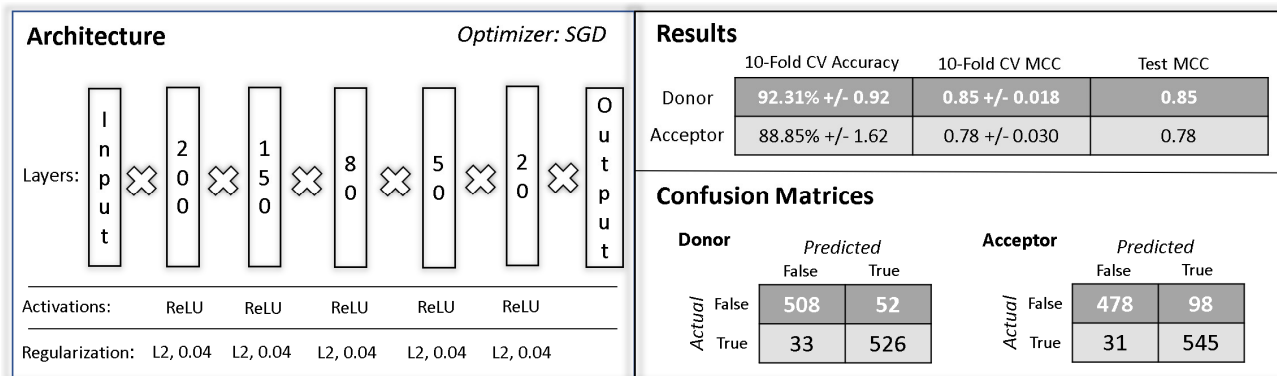
**Figure 10:** Hyperparameters and results for model 3. See figure 8 for description of common elements.

The model was defined with Keras using the code snippet below:

```
1. def buildModelAndCompile():
2.     K.clear_session()
3.     model = Sequential()
4.     model.add(Flatten(input_shape=(140, 4)))
5.     model.add(Dense(200, activation = 'relu'))
6.     model.add(Dense(150, activation = 'relu'))
7.     model.add(Dense(80, activation = 'relu'))
8.     model.add(Dense(50, activation = 'relu'))
9.     model.add(Dense(20, activation = 'relu'))
10.    model.add(Dense(2, activation = 'softmax'))
11.    model.compile(optimizer = 'sgd', loss = 'binary_crossentropy', metrics=['accuracy'])
12.    return(model)
```

### Model 4 – Five ReLU Hidden Layers with L2 Regularization

Model 4 is similar to model 3 but with L2 regularization added to each layer. The lambda value was after several rounds of optimization set to 0.04, and any further increase lead to untrainable models giving results akin to model 2. **Figure 11** illustrates the model and its results.



**Figure 11:** Hyperparameters and results for model 4. See figure 8 for description of common elements. In the Architecture box, the “L2, 0.04” elements in the regularization section represents L2 regularization with the lambda parameter set to 0.04.

The model was defined with Keras using the code snippet below:

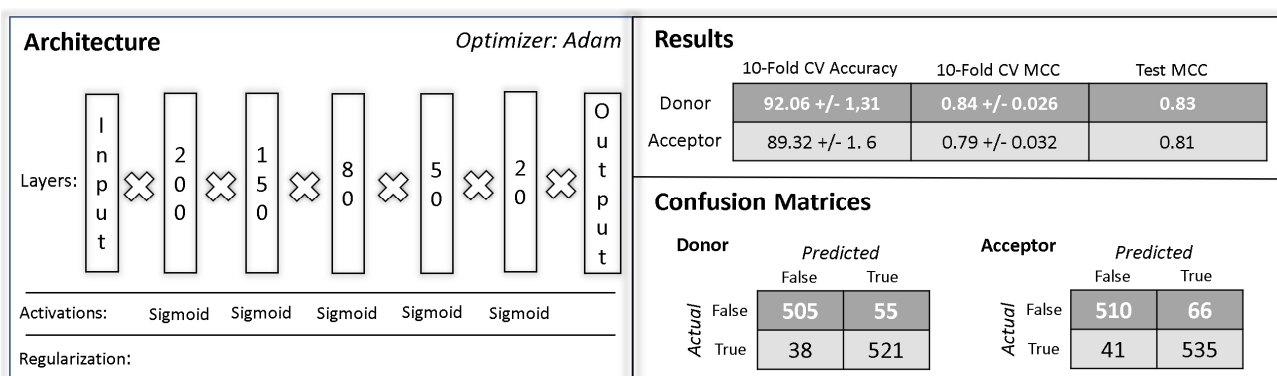
```

1. def buildModelAndCompile():
2.     K.clear_session()
3.     model = Sequential()
4.     model.add(Flatten(input_shape=(140, 4)))
5.     model.add(Dense(200, activation = 'relu', kernel_regularizer=regularizers.l2(0.04)))
6.     model.add(Dense(150, activation = 'relu', kernel_regularizer=regularizers.l2(0.04)))
7.     model.add(Dense(80, activation = 'relu', kernel_regularizer=regularizers.l2(0.04)))
8.     model.add(Dense(50, activation = 'relu', kernel_regularizer=regularizers.l2(0.04)))
9.     model.add(Dense(20, activation = 'relu', kernel_regularizer=regularizers.l2(0.04)))
10.    model.add(Dense(2, activation = 'softmax'))
11.    model.compile(optimizer = 'sgd', loss = 'binary_crossentropy', metrics=['accuracy'])
12.    return(model)

```

### Model 5 – Five Hidden Layers with the Adam Optimizer

Model 5 is a carbon copy of Model 2 with the exception that the Adam optimizer was used instead of SGD. **Figure 12** shown below represents the model.



**Figure 12:** Hyperparameters and results for model 5. See figure 8 for description of common elements.

The model was defined with Keras using the code snippet below:

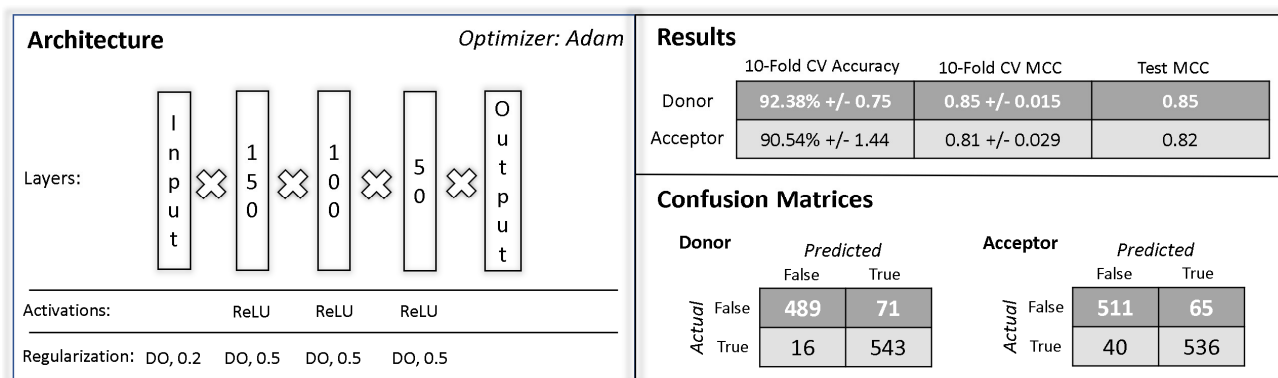
```

1. def buildModelAndCompile():
2.     K.clear_session()
3.     model = Sequential()
4.     model.add(Flatten(input_shape=(140, 4)))
5.     model.add(Dense(200, activation = 'sigmoid'))
6.     model.add(Dense(150, activation = 'sigmoid'))
7.     model.add(Dense(80, activation = 'sigmoid'))
8.     model.add(Dense(50, activation = 'sigmoid'))
9.     model.add(Dense(20, activation = 'sigmoid'))
10.    model.add(Dense(2, activation = 'softmax'))
11.    model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics=['accuracy'])
12.    return(model)

```

### Model 6 – Three ReLU Hidden Layers with Dropout Regularization and the Adam Optimizer

Model 6 is trained with the Adam optimizer and consists of three hidden layers with ReLU activation and dropout. **Figure 13** depicts the model and the associated results.



**Figure 13:** Hyperparameters and results for model 6. See figure 8 for description of common elements. In the Regularization section of the Architecture box, DO stands for dropout and the number associated with it indicates the drop out rate. For example, 20% of the inputs are randomly dropped in each mini-batch when training this model.

The model was defined with Keras using the code snippet below:

```

1. def buildModelAndCompile():
2.     K.clear_session()
3.     model = Sequential()
4.     model.add(Flatten(input_shape=(140, 4)))
5.     model.add(Dropout(0.20))
6.     model.add(Dense(150, activation = 'relu'))
7.     model.add(Dropout(0.50))
8.     model.add(Dense(100, activation = 'relu'))
9.     model.add(Dropout(0.50))
10.    model.add(Dense(50, activation = 'relu'))
11.    model.add(Dropout(0.50))
12.    model.add(Dense(2, activation = 'softmax'))
13.    model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics=['accuracy'])
14.    return(model)

```

# Discussion

## Results in Comparison to Recent Literature

The results of the various models are summed up in **Figure 14**, where only the test performance is listed. The standard deviations reported in the previous section were found using a single round of 10-fold cross validation, which means they are perhaps somewhat large comparatively speaking. This could be remedied by repeating the 10-fold cross validation several times, but due to the additional run time cost, already extensive due to limited processing power availability, the single run approach was deemed sufficient. In any case, all the test results fell within one standard deviation of the 10-fold CV estimate, which indicates that none of the test MCC scores were improbable outliers. As such, the test MCC, arguably the superior performance metric, will be used in the comparison to other models from recent literature.

Model	Donor Test MCC	Acceptor Test MCC
Model 1 – Basic One Hidden Layer	0.85	0.78
Model 2 – Basic Five Hidden Layers	NA	NA
Model 3 – Five ReLU Hidden Layers	0.80	0.75
Model 4 – Five ReLU/L2 Hidden Layers	0.85	0.78
Model 5 – Basic Five Hidden Layers Adam	0.83	0.81
Model 6 – Three Hidden Layers Dropout	0.85	0.82

**Figure 14:** Test MCC results for models 1-6. NA stands for Not Available, which reflects the model's inability to learn from the dataset. See the Vanishing Gradient subsection for a full explanation.

Aside from Model 2, all the tested models have test MCC scores around 0.80, but a quick evaluation reveals that Model 6, using the combination of ReLU activation, the Adam optimizer, and drop-out for regularization, is the one that performs the best, though only by a small margin. Model 2's failure to produce MCC scores is discussed in the Vanishing Gradient subsection.

For comparison, the results of Pashaei Et Al. (2016) and Naito (2018) will be used [7, 9]. Both papers utilized the HS<sup>3</sup>D dataset, using a 1:1 ratio between true and false samples, while also reporting the performance of their models in MCC, though both only use 10-fold cross validation without any test set. Pashaei Et Al. proposed model was a combination of a first-order Markov



model for feature selection with a Random forest classifier for prediction, but they also report the performance of other machine learning models. Naito built a relatively complex neural network consisting of several convolutional layers, a recurrent bidirectional middle layer, and a fully-connected final layer. See **figure 15** for a summary of the results for the 1:1 dataset. Both papers use 5 rounds of 10-fold cross validation to produce their performance estimates.

Model	Donor MCC	Acceptor MCC
First Order Markov Model with SVM classifier	0.85	0.77
Linear SVM with Bayes kernel	0.85	0.79
First Order Markov Model with Random Forest classifier	0.86	0.81
Convolutional NN with Recurrent Layer	0.93	0.90

**Figure 15:** MCC results from multiple rounds of 10-fold cross validation for models from Pashaei *Et Al.* (2016) and Naito (2018), listed here without standard deviations for clarity and because they were all under 0.004. SVM stands for Support Vector Model.

There is no room for the discussion of the particulars of the individual models used for comparison, but it's safe to say that the first three listed in figure 15 represent fairly sophisticated machine learning models that make use of some of the most prominent machine learning algorithms, such as Random Forest and Bayes kernel. The best performing of these more traditional approaches is the First Order Markov Model with the Random Forest classifier, having MCC scores of 0.86 and 0.81 for the donor and acceptor sites, respectively. This is fairly similar to the performance of Model 6, which has corresponding test scores of 0.85 and 0.82. It is a testament to the effectiveness of Tensorflow with the Keras wrapper and perhaps the power of neural networks in general that the limited optimization of a relatively simplistic NN can result in a model that can compete with a recently published scientific paper on a classification task as relevant as splice site prediction.

On the other hand, it becomes clear that the full capacity of neural networks hasn't been realized in Model 6 when compared to the convolutional NN presented by Naito (2018). The MCC scores are markedly higher, 0.93 and 0.90, clearly showcasing that the more advanced NN model was able to capture some patterns in the data that lead to more robust predictions. It is perhaps not surprising that a feed-forward network utilizing convolutional layers, which has demonstrated incredible effectiveness in image recognition and similar tasks, is able to outcompete other models

when analyzing what is essentially pictures 140 pixels long and 4 pixels wide, but making any statements on the matter is marred with uncertainty, which will be elaborated on in the final subsection. While the performance of the tested models of this paper is not quite on the level of the convolution neural network, their performance seems surprisingly adequate compared to the more traditional ML approaches.

### **Model Capacity**

At its most basic, the model capacity of a neural network can be increased either by inserting more neurons into a given layer, or by adding additional layers. As there is no standardized and well-understood accurate metric for the extent to which a model's capacity is increased by these modifications, the estimate of neural network model capacities is often based on heuristics or the number of trainable parameters, the latter being a gross simplification [19, 32]. To exemplify this, consider that Model 1, a one-hidden-layer model with 500 neurons, has 281,502 trainable parameters, and that Model 6, showing similar performance but with 3 hidden layers and a total of 300 neurons, only has 104,402 trainable parameters. A great amount of regularization was added to Model 6 as well, in order to prevent overfitting. These two observations suggest that adding additional layers to a neural network increases model capacity greatly, and that the number of trainable parameters might be a very flawed estimate. Arguably, the experiments in this paper wasn't set up to systematically test how model modifications affect the capacity, and as such it is very possible that both Model 1 and Model 6 has capacity in abundance for this particular classification task, and as such it is hard to make deductions with any certainty. If both models were reduced in neuron count until they started to show substantial loss in their performance metrics, then there would a better case to be made that they both represent models with just enough model capacity to capture the information in the dataset, allowing for more fair comparison. Admittedly, that would still be a flawed heuristically-based approach. In recent years, some researchers have attempted to combat the use of both heuristics and simple estimates based on trainable parameters by developing advanced mathematical metrics for model capacity, such as "total valid bits" [32] and "Algebraic topology" [33], which is beyond the scope of this paper to delve into. Suffice it to say that various robust estimates of model capacity will likely become standard to employ in the future as the field of neural networks continue to develop.

## **Overfitting**

Model 3 has 5 layers and a sum of 500 neurons, which should translate into a relatively high model capacity and thus, a high risk of overfitting the model to the training data. However, without any kind of regularization, it achieves a test MCC of 0.80 for donors and 0.75 for acceptor. Model 4, which is the same model but with L2 regularization added to each hidden layer, arguably seems to have better generalizability with its test MCC of 0.85 for donors and 0.78 for acceptors, but the difference between the two models is not very pronounced. One effect that in part can account for this lack of difference between the regularized and unregularized versions, and to some extent also explain how the similarly completely unregularized Model 1 can perform well, is the poorly understood high intrinsic generalization capacity of neural networks [34], which is possibly related to the implicit regularization attribute of the SGD algorithm [35] and its variants. However, what is likely to be of much higher relevance is that all the models were trained using only 40 epochs, which in and of itself could be argued to be a type of implicit regularization. The difference between Model 3 and Model 4 is likely to become far more marked if both were instead trained for 400 epochs instead, for example. A more orderly way to attain an implicit regularization effect similar to limiting the number of epochs would be to use early stopping, which entails stopping the training process if the loss on the validation set hasn't decreased for a set number of epochs. Model 6 is difficult to compare with the other architectures, but its aggressive drop-out rates reflects that a lot of regularization was found necessary in the optimization process to limit the effect of overfitting. Since Model 6 differs in multiple hyperparameters from the other models, it cannot be stated with any confidence that the drop out regularization is the cause of its superior performance, but since drop-out has become a popular main-stain in NN hyperparameter searches due to its effectiveness, it is not completely unlikely that it is in part what made Model 6 the best performing of the lot.

## **Vanishing Gradients**

Model 2 fails to generate MCC scores in both the cross-validation and test settings. The reason reveals itself in the two confusion matrices, which show that the model labels every sample as not being a splice site. This means that both the true positive and false positive categories are 0, thus leading to a denominator of 0 in the MCC calculation. This failure to properly train the model is most likely a demonstration of the aforementioned issue of vanishing gradients encountered

when using activation functions such as Sigmoid or Hyperbolic Tangent, which both have limited output space and thus lead to relatively small derivatives. Model 1, having a test MCC of 0.85 for donor sites and 0.78 for acceptors, is not hampered by this potential problem, which is to be expected given that it has only one single hidden layer. Model 2 on the other hand, has 5 hidden layers, allowing for the vanishing gradients phenomenon to manifest, leading to the weights and biases of the first layers becoming more or less untrainable due to a miniscule gradient. Thus, Model 2 ends up merely designating every observation as the same class, failing in the training process to find a way to a local minimum better than that naïve model. What strongly suggests that vanishing gradients is the cause of this is that Model 3, being a carbon-copy of Model 2 with the exception of using ReLu activation instead of Sigmoid, manages to achieve MCC scores around 0.8, clearly showing that it was possible to meaningfully train its parameters.

Model 5 also has hyperparameters similar to Model 2, but it uses the Adam optimizer instead of SGD. Interestingly, this model also manages to reach a better local minimum in the training process compared to Model 2, achieving a test MCC of 0.83 for donors and 0.81 for acceptors. While it is beyond this report to delve into the details of the algorithmic components that might be responsible for this improved ability to learn from the training data despite potential issues of vanishing gradients, it is safe to say that the result of model 5 showcase why the Adam optimizer has become the preferred choice.

Vanishing gradients is just one side of the coin of what is generally called unstable gradients. The other side is exploding gradients [36], which occurs when the loss function gradient grows exponentially larger for each successive layer in the neural network, leading to far too extreme weight and bias updates, whereby the training process is sabotaged. No instance of this phenomenon is clearly expressed in the models presented in this report, but the effect might at some level be present but undetected, as it can best be seen by observing the size of the actual weights and biases, which wasn't within the scope of this report or the optimization process.

### **Hyperparameter Search and Performance**

While it might be tempting to claim that it is not possible for relatively simple feed-forward neural network models without convolutional layers to perform as well on the splice site prediction task as the far more involved convolution neural network used as a yardstick in this report, it is folly to

do so. Model 1 and Model 6 with their highly dissimilar hyperparameters but similar results reveal that at least some effort has been put into exploring the hyperparameter space, but that still leaves open the very real possibility that some arcane configuration of a non-convolutional feed-forward neural network would be able to match or even outcompete the convolution neural network. This underlines one of the most critical downfalls of neural networks in general, the fact that the search for the ideal hyperparameters can be very time-consuming and every proposed optimal design will be plagued by uncertainty as it is impossible to refute the possibility that an even better version exists. This predicament is mainly due to the fact that the interactions between the countless hyperparameters and any given data set is, for now, far beyond our complete theoretical understanding, thus rendering optimization of neural networks in large part an art of trial and error with great reliance on heuristics [23, 37].

The difficulty of searching for optimal hyperparameters can to some extent be mitigated by the use of automatic hyperparameter optimization in the form of grid search or random search, but these options further compound another key issue with neural networks, that being the computational cost of training them.

## Conclusion

By using the high-level wrapper Keras in conjunction with the Tensorflow library this paper has relatively effortlessly implemented a number of feed-forward neural networks that predict the presence of splice sites in human DNA sequences 140 bases long after having been trained on the HS<sup>3</sup>D dataset. The models, upon inspection, served to illustrate the concepts of model capacity, overfitting and vanishing gradients and how they relate to the various hyperparameters chosen for this paper. Furthermore, the implemented models were evaluated using 10-fold cross validation and a lock box approach. Ultimately the performance was measured using the balanced Matthew's Correlation Coefficient metric, with the best model having MCC test scores of 0.85 and 0.82 when predicting acceptor and donor splice site, respectively. When the performance was compared to other machine learning models in recent literature, the ones implemented in this paper proved competitive, though outperformed by a more advanced neural network.

# Bibliography

1. Zhang, M.Q., *Statistical features of human exons and their flanking regions*. Human Molecular Genetics, 1998. **7**(5): p. 919-932.
2. De Conti, L., M. Baralle, and E. Buratti, *Exon and intron definition in pre-mRNA splicing*. Wiley Interdiscip Rev RNA, 2013. **4**(1): p. 49-60.
3. Matlin, A.J., F. Clark, and C.W. Smith, *Understanding alternative splicing: towards a cellular code*. Nat Rev Mol Cell Biol, 2005. **6**(5): p. 386-98.
4. Chen, T.M., C.C. Lu, and W.H. Li, *Prediction of splice sites with dependency graphs and their expanded bayesian networks*. Bioinformatics, 2005. **21**(4): p. 471-482.
5. Sonnenburg, S., et al., *Accurate splice site prediction using support vector machines*. BMC Bioinformatics, 2007. **8**.
6. Zhang, Q.W., et al., *Splice sites prediction of Human genome using length-variable Markov model and feature selection*. Expert Systems with Applications, 2010. **37**(4): p. 2771-2782.
7. Pashaei, E., M. Ozen, and N. Aydin, *Random Forest in Splice Site Prediction of Human Genome*. Xiv Mediterranean Conference on Medical and Biological Engineering and Computing 2016, 2016. **57**: p. 512-517.
8. Min, S., B. Lee, and S. Yoon, *Deep learning in bioinformatics*. Brief Bioinform, 2017. **18**(5): p. 851-869.
9. Naito, T., *Human Splice-Site Prediction with Deep Neural Networks*. Journal of Computational Biology, 2018. **25**(8): p. 954-961.
10. Zhang, Y., et al., *DeepSplice: Deep Classification of Novel Splice Junctions Revealed by RNA-seq*. 2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), 2016: p. 330-333.
11. Jaganathan, K., et al., *Predicting Splicing from Primary Sequence with Deep Learning*. Cell, 2019. **176**(3): p. 535-548 e24.
12. Petersen, S.B., et al., *Training neural networks to analyse biological sequences*. Trends Biotechnol, 1990. **8**(11): p. 304-8.
13. Brunak, S., J. Engelbrecht, and S. Knudsen, *Prediction of human mRNA donor and acceptor sites from the DNA sequence*. J Mol Biol, 1991. **220**(1): p. 49-65.
14. Pollastro, P. and S. Rampone, *(HSD)-D-3, a dataset of Homo Sapiens Splice regions, and its extraction procedure from a major public database*. International Journal of Modern Physics C, 2002. **13**(8): p. 1105-1117.
15. Skocik, M., et al., *I TRIED A BUNCH OF THINGS: THE DANGERS OF UNEXPECTED OVERFITTING IN CLASSIFICATION*. 2016: p. 078816.
16. Matthews, B.W., *Comparison of the predicted and observed secondary structure of T4 phage lysozyme*. Biochim Biophys Acta, 1975. **405**(2): p. 442-51.
17. Chicco, D., *Ten quick tips for machine learning in computational biology*. BioData mining, 2017. **10**(1): p. 35.
18. Glorot, X. and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, T. Yee Whye and T. Mike, Editors. 2010, PMLR %J Proceedings of Machine Learning Research: Proceedings of Machine Learning Research. p. 249--256.
19. Goodfellow, I., Y. Bengio, and A. Courville, *Deep learning*. 2016: MIT press.
20. Glorot, X., A. Bordes, and Y. Bengio, *Deep Sparse Rectifier Neural Networks*, in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, G. Geoffrey, D. David, and D. Miroslav, Editors. 2011, PMLR %J Proceedings of Machine Learning Research: Proceedings of Machine Learning Research. p. 315--323.

21. Clevert, D.-A., T. Unterthiner, and S.J.a.p.a. Hochreiter, *Fast and accurate deep network learning by exponential linear units (elus)*. 2015.
22. Manessi, F. and A. Rozza, *Learning Combinations of Activation Functions*. 2018 24th International Conference on Pattern Recognition (Icpr), 2018: p. 61-66.
23. Nielsen, M.A., *Neural networks and deep learning*. Vol. 25. 2015: Determination press USA.
24. Bottou, L., F.E. Curtis, and J. Nocedal, *Optimization Methods for Large-Scale Machine Learning*. Siam Review, 2018. **60**(2): p. 223-311.
25. Rumelhart, D.E., G.E. Hinton, and R.J. Williams, *Learning Representations by Back-Propagating Errors*. Nature, 1986. **323**(6088): p. 533-536.
26. Kingma, D.P. and J. Ba, *Adam: A method for stochastic optimization*. arXiv preprint arXiv:1412.6980, 2014.
27. Ruder, S., *An overview of gradient descent optimization algorithms*. arXiv preprint arXiv:1609.04747, 2016.
28. Dozat, T., *Incorporating nesterov momentum into adam*. 2016.
29. Hinton, G.E., et al., *Improving neural networks by preventing co-adaptation of feature detectors*. arXiv preprint arXiv:1207.0580, 2012.
30. Krogh, A. and J.A. Hertz, *A Simple Weight Decay Can Improve Generalization*. Advances in Neural Information Processing Systems 4, 1992. **4**: p. 950-957.
31. Srivastava, N., et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Journal of Machine Learning Research, 2014. **15**: p. 1929-1958.
32. Wang, A., et al. *Deep Neural Network Capacity*. arXiv e-prints, 2017.
33. Guss, W.H. and R. Salakhutdinov, *On characterizing the capacity of neural networks using algebraic topology*. arXiv preprint arXiv:1802.04443, 2018.
34. Zhang, C., et al., *Understanding deep learning requires rethinking generalization*. arXiv preprint arXiv:1611.03530, 2016.
35. Lei, D., et al., *Implicit Regularization of Stochastic Gradient Descent in Natural Language Processing: Observations and Implications*. arXiv preprint arXiv:1811.00659, 2018.
36. Pascanu, R., T. Mikolov, and Y. Bengio. *On the difficulty of training recurrent neural networks*. in *International conference on machine learning*. 2013.
37. LeCun, Y., Y. Bengio, and G. Hinton, *Deep learning*. Nature, 2015. **521**(7553): p. 436-44.