Master thesis : Feature selection with deep neural networks

Auteur : Vecoven, Nicolas
Promoteur(s) : Geurts, Pierre
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"
Année académique : 2016-2017
URI/URL : http://hdl.handle.net/2268.2/2552

University of Liege

Faculty of Applied Sciences

Feature selection with deep neural networks

Author:
Nicolas Vecoven

Supervisor:
Prof. Pierre Geurts

Master thesis submitted for the degree of
MSc in Computer Science and Engineering

Jury members : Prof. Damien Ernst, Prof. Pierre Geurts, Prof. Louis Wehenkel, Prof. Laurent Mathy

Academic year 2016-2017

Abstract

Feature selection with deep neural networks

by Nicolas Vecoven *supervised by Prof. Pierre Geurts*

Variable and feature selection have become the focus of much research, especially in bioinformatics where there are many applications. Machine learning is a powerful tool to select features, however not all machine learning algorithms are on an equal footing when it comes to feature selection. Indeed, many methods have been proposed to carry out feature selection with random forests, which makes them the current go-to model in bioinformatics.

On the other hand, thanks to the so-called deep learning, neural networks have benefited a huge interest resurgence in the past few years. However neural networks are blackbox models and very few attempts have been made in order to analyse the underlying process. Indeed, quite a few articles can be found about feature extraction with neural networks (for which the underlying inputs-outputs process does not need to be understood), while very few tackle feature selection.

In this document, we propose new algorithms in order to carry out feature selection with deep neural networks. To assess our results, we generate regression and classification problems which allow us to compare each algorithm on multiple fronts: performances, computation time and constraints. The results obtained are really promising since we manage to achieve our goal by surpassing (or equaling) random forests performances in every case (which was set to be our state-of-the-art comparison).

Due to the promising results obtained on artificial datasets we also tackle the DREAM4 challenge. Due to the very small number of samples available in the datasets, this challenge is supposedly an ill-suited problem for neural networks. We were nevertheless able to achieve near state of the art results.

Finally, extensions are given for most of our methods. Indeed, the algorithms discussed are very modulable and can be adapted regarding the problem faced. For example, we explain how one of our algorithm can be adapted in order to prune neural networks without losing accuracy.

Acknowledgments

First I would like to thank Prof. Pierre Geurts for the time he spent working with me as well as his availability. It makes no doubt that the results of this thesis would not have been the same without his many advices. I am also grateful to him for allowing me to work on such an interesting topic.

I would then like to thank Jean-Michel Begon for his availability as well as for the useful links he provided me to guide my research.

I would also like to thank vân anh huynh-thu for proofreading this document.

Finally I would like to address my gratitude to my friends and family for their support. Especially, to Frederic who accepted to proofread this document.

Liège, May 2017

Contents

Chapter 1

Introduction

Variable and feature selection have become the focus of much research, especially in bioinformatics where there are many applications. Machine learning is a powerful tool to select features, however not all machine learning algorithms are on an equal footing when it comes to feature selection. Indeed, many methods have been proposed to carry out feature selection with random forests, which makes them the current go-to model in bioinformatics. This mainly comes from the fact that random forests are well known to be good “out-of-the-bag” algorithms and they do not need huge amount of data in order to achieve good results.

On the other hand, thanks to the so-called deep learning, neural networks have benefited a huge interest resurgence in the past few years. However neural networks are blackbox models and very few attempts have been made in order to analyse the underlying process. Indeed, quite a few articles can be found about feature extraction with neural networks (for which the underlying inputs-outputs process does not need to be understood), while very few tackle feature selection. Furthermore, neural networks are known to require lots of data and computation time in order to achieve good performances. Since data are often hard to obtain in bioinformatics, this is already a burden for neural networks. Nevertheless, some attempts were made to select features using neural network, unfortunately most of them used very shallow networks and others were directed to very specific datasets.

In this document, we propose to address that missing part for neural networks and give a general way to carry out feature selection using deep neural networks. To this end, we will review the existing techniques and give our assessment method in Chapter 2. In Chapter 3, we will introduce and detail our algorithms and show some results on artificial datasets. Finally in Chapter 4, we use the most robust of our methods to tackle the DREAM4 challenge and compare our results with the state-of-the-art methods. We consider this as a good stress-test since the challenge consists of datasets with very few data samples and should a priori handicap neural networks, especially deeper ones.

Chapter 2

Overview of deep neural networks and motivation for feature selection

This chapter first aims at introducing feature selection, to this end Section 2.1 will explain briefly what it is and how to assess performances. Section 2.2 will then introduce some of the existing and most often used feature selection methods. In Subsection 2.3.2 we will detail the current state of deep learning and give the constraints which we put on our neural networks throughout the document. Subsection 2.3.3 reviews some of the methods that were previously proposed in order to carry out feature selection with neural networks. Finally, in Subsection 2.3.4 we motivate the methods that will be proposed in Chapter 3.

2.1 Feature selection and its assessment

2.1.1 What is it ?

Before going into what feature selection is into details, let us introduce a broader concept of machine learning. Indeed, to understand what is going to follow, one needs to understand what supervised learning is. It can be defined as a machine learning task that infers a function from labeled data. To this end, a model is learned on training data which consists of inputs-outputs pairs. The goal is to learn a (potentially non-linear) mapping function from inputs to outputs, such as to minimise the error while mapping unseen examples (i.e. that don't belong to the training data). The optimal scenario would be for the model to map new inputs to the exact outputs. This implies that the learning algorithm must generalise from the training data in order to handle unseen inputs correctly. Supervised learning has plenty of applications and is often used for computer vision, speech recognition, time-series predictions and others. In the next paragraph we see how supervised learning can also be used to carry out feature selection.

Feature selection (also known as variable selection or attribute selection) is the process of selecting a subset of relevant features (variables, predictors) which we are going to define. Let X be the set of n input variables and Y be the output.

Definition 1. *A variable $V \in X$ is relevant iff there exists a subset $B \in X$ such that $V \not\perp Y|B$. A variable is irrelevant if it is not relevant.*

Relevant features can be further divided into strongly relevant and weakly relevant categories:

Definition 2. *A Variable V is strongly relevant iff $Y \not\perp V|X \setminus V$. A variable V is weakly relevant if it is relevant but not strongly relevant.*

Feature selection can be associated with the two following problems ([Nilsson et al., 2007]):

- **All-relevant problem.** Which consists in finding all relevant features.
- **Minimal optimal problem.** Which consists in finding a subset $M \subseteq X$ such that $Y \perp X \setminus M|M$ and such that no proper subset of M satisfies this property.

In other words, the *minimal optimal problem* consists in finding the smallest subset of input variables which contains the whole information (i.e. such that no other input variable could be added to bring information) while the *all-relevant problem* aims at finding all the variables that bring information regardless that the same information might be brought by multiple inputs. Feature selection is often done by associating importances to input dimensions of a model. Thus the higher the importance of an input feature, the more informative (i.e. relevant) it is likely to be. Different methods exist to carry out feature selection and belong to one of the three following categories:

- **Filter methods.** These are not based on machine learning algorithms, and use general features such as correlation with the variable(s) to predict.
- **Wrapper methods.** These methods use a trained model and some test samples in order to derive feature importances from performances regardless the type of machine learning algorithm used.
- **Embedded methods.** These use internal parameters of the machine learning algorithms. A good example of this is the splitting criteria in decision trees, which is a good measure of feature importance (this idea will be detailed in Section 2.2). This document will mostly focus on such methods.

Algorithm I gives the main steps to be conducted when using embedded methods.

Algorithm I. *An overview of the high-level steps to be conducted in order to carry out feature selection with an embedded method.*

1. Choose a learning model.
2. Train this model on a given dataset.
3. Analyse the model parameters in order to obtain input importances.

This method is often used in areas of applications for which datasets have high dimensional input space (with hundreds or thousands of variables available). These include text processing of internet documents, gene expression array analysis and combinational chemistry. These methods aim at improving the performances of predictors by reducing overfitting, having a better understanding of the underlying distribution and avoid curse of dimensionality. A deeper understanding of what feature selection is can be obtained by reading Guyon Isabelle's paper [Guyon and Elisseeff, 2003]. As a motivation for feature selection, Figure 2.1 shows the performance degradation of a deep neural network while adding noise. The noise consists of adding multiple irrelevant features generated using different noise types (generated by scikit-learn [skl, b]) to the inputs, while the relevant inputs are left as such. As can be seen, it would be very helpful to be able to select the useful features and retrain a network only using relevant features. In this particular case, it would lead to an accuracy of around 95%, increased from around 60%. The next paragraph aims at setting our goal as well as defining the measure of success that will be used throughout this document.

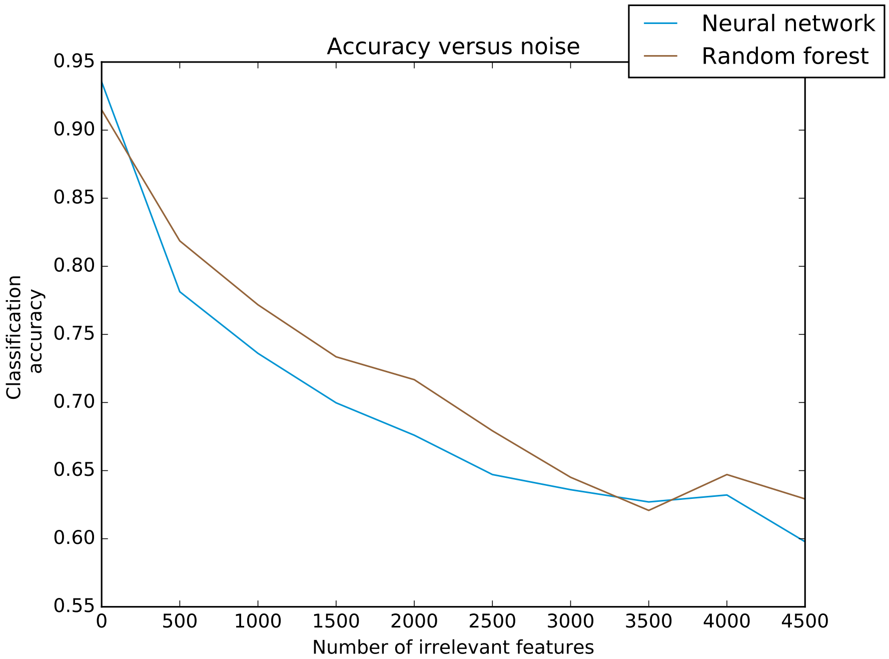

Figure 2.1: Effect of noise on accuracy performances of deep neural networks and random forests. These results were obtained on a binary classification problem, generated with the Sklearn hypercube data generation [skl, b] using 25 informative variables. The neural network is composed of 4 hidden layers of 500 neurons each and the random forest consists of 5000 trees trained with $K = log_2(p)$ where p is the number of input features. Each model has been learned on 2000 training samples and the performances evaluated on 5000 validation samples. More information on the dataset is given in Appendix A

2.1.2 How to assess it ?

The goal of this document is to propose and analyse algorithms to carry out feature ranking with deep neural networks, i.e. giving a measure of usefulness for each input variable. To this end, a performance measure has to be defined. Let us start by introducing confusion matrices which are often built in order to assess the performances of classification models. Consider a binary classification problem with a class corresponding to a positive outcome (for example an alarm activation) and the other to a negative outcome (the alarm doesn't activate). Also consider a binary classification model which is used to classify an input (for example a motion detector) to one of the two classes. For each data sample, the classification made by the model belongs to one of the following categories:

- **True positive (TP).** This occurs if the model activates the alarm when it should have been. No error is made.

- **True negative (TN).** This occurs if the model doesn't activate the alarm rightfully so (i.e. the alarm should not have been activated). No error is made.

- **False positive (FP).** This occurs when the model activates the alarm although it should not have been. An error is made and leads to *Type 1 error.*

- **False negative (FN).** This occurs when the model doesn't activate the alarm although it should have been. An error is made and leads to *Type 2 error.*

The confusion matrix is built as shown on Table 2.1:

		Ground truth	
		Positive	Negative
Model output	Positive	TP	FP
	Negative	FN	TN

Table 2.1: Confusion matrix

From the confusion matrix, two measures can be introduced:

- **Precision.** Which is defined as $\frac{TP}{TP+FP}$ and represents the proportion of true positives retrieved amongst all classified positives.

- **Recall.** Which is defined as $\frac{TP}{TP+FN}$ and represents the proportion of true positives retrieved amongst all the real positives.

These measures can also be used in order to assess feature selection. Indeed, we will use datasets for which we know the informative and non-informative variables (due to how we generate them) such that we have a ground truth. Algorithm II details how we use the notion of precision-recall in our particular case and defines the "*area under the precision-recall curve*" (AUPR), which will be used as our assessment measure.

Algorithm II. *An algorithm to compute the AUPR based on feature importances.* ***imp[1,...,n]*** *is the sorted (descending order) vector of importances that has been computed, i.e. we have* $imp[1] = i$ *where i is the index of the input which has the biggest importance.* ***informative[...]*** *is the vector containing the indices of informative variables (ground truth). Finally, let us define the function* $\mathbb{1}(x)$*, we have*

$$\mathbb{1}(x) = \begin{cases} 1, & \textit{if } x \textit{ is true} \\ 0, & \textit{else} \end{cases}$$

```
1. Compute the points of the precision-recall curve.
    (a) For $t$ in [1,...,n]:
        i. TP (true positives) ← $\sum_{i=1}^{t} \mathbb{1}(imp[i] \in informative)$
        ii. FP (false positives) ← $\sum_{i=1}^{t} \mathbb{1}(imp[i] \notin informative)$
        iii. FN (false negative) ← $\sum_{i=t+1}^{n} \mathbb{1}(imp[i] \in informative)$
        iv. $precision_t \leftarrow \frac{TP}{TP+FP}$
        v. $recall_t \leftarrow \frac{TP}{TP+FN}$
        vi. Create point with the pair $[precision_t, recall_t]$
    (b) Build the curve with each point.
2. Compute the area under the curve.
    (a) AUPR = 0
    (b) $precision_0 \leftarrow 1$
    (c) $recall_0 \leftarrow 0$
    (d) For $i$ in [0,...,n-1]:
        i. $base \leftarrow precision_{i+1} + precision_i$
        ii. $height \leftarrow \frac{recall_{i+1} - recall_i}{2}$
        iii. $AUPR \leftarrow AUPR + (base * height)$
```

Algorithm II gives an AUPR equal to one if and only if none of the non-informative variable has a higher importance than an informative one. Inversely, the AUPR will be very close to zero if no informative variable has a higher importance than a non-informative one. All in all, this is a great measure to see if the feature selection is made correctly, the higher the AUPR the better the distinction between informative and non-informative variables. In this document, we will mostly limit ourselves to distinguishing informative from non informative variables, indeed, we will not look at the ranking done between informative variables. Although we will make an exception in Section 3.5.1.4 to discuss the effect of redundant variables.

2.2 Literature review

Since deep neural networks are considered as blackbox model, they have never been much used for feature selection. However few attempts have been made and will be described in the next section. Unfortunately, they have never been as popular as other techniques. We will now talk briefly about what algorithms are often used for feature selection and considered as giving some of the best results. Note that we limit ourselves to the embedded algorithms, this means that we will not talk about algorithms that do not use a learning model in order to achieve feature selection. An overview and comparison of many of the feature selection algorithms is well detailed in [Saeys et al., 2007]. In the next paragraph, we chose to focus on a few of them.

Multiple models can be used to achieve feature selection, for example:

- Linear regression with regularisation method (such as LASSO and elastic net [Zou and Hastie, 2005])
- Weight vector of SVM ([Guyon et al., 2002]).
- Random forests.

Random forests are some of the easiest models to use and work well “out of the box” for a wide range of problems. Therefore, they are often chosen for feature selection tasks due to their ease of use as well as their performances. They are especially used in bioinformatics for a multitude of applications (some of which are detailed in [Qi, 2012]). There exist multiple algorithms to compute feature importances using random forests ([Qi, 2012]):

- **Permutation based variable importance.** This algorithm consists in permuting the variables at test time and looking at the accuracy loss. This technique is part of the wrapper algorithms, it can thus be used with any learning algorithm. Therefore it will be looked at for deep neural network in Chapter 3.

- **Gini importance.** This method is directly based on the Gini index which measures the level of impurity / inequality of the samples assigned to a node based on a split at its parent. For instance, under the binary classification case, let p represent the proportion of class 1 samples assigned to a certain node n and $1 - p$ as the proportion of class 2 samples. The Gini index at n is defined as:

  $$G_n = 2p(1 - p)$$

  The Gini importance value of a feature in a single tree is then defined as the sum of the Gini index reduction (from parent to children) over all nodes in which the specific feature is used to split. The overall importance in the forest is defined as the sum or the average of its importance value among all trees in the forest.

- **Revised RF feature importance.** There are many other feature importance measures that have been created. One of them is called the depth importance measure and reflects the quality of the node splits. It is in fact similar to the Gini importance, but the difference is that the depth importance takes into account the position of the node in the trees. This formula has been proposed by [Chen et al., 2007]. Some other techniques (such as [Altmann et al., 2010], [Strobl et al., 2008], [Wang et al., 2010]) have been proposed but will not be detailed here.

Overall, we decided to use random forests with the Gini importance metric (as implemented in Scikit-learn [skl, a]) as our “state-of-the-art” algorithm. Our goal is thus to show that we are able to achieve similar or better performances on a wide variety of tasks using our proposed algorithms with deep neural networks.

2.3 Artificial neural networks

2.3.1 Introduction

Neural networks were first created in an attempt to model the human brain. However, throughout the years, they have become much different of what they used to be and derived from their initial purpose. They are now used as powerful machine learning tools in a multitude of fields: computer vision, stock market analysis and robotics being some of them. This subsection aims at giving a brief introduction on their internal working such that the rest of the document can be understood.

First, let's look at the structure of neural networks. They are divided in multiple layers, each of which contains multiple neurons. Each neuron is connected to the following layer through weights, as shown on Figure 2.2.

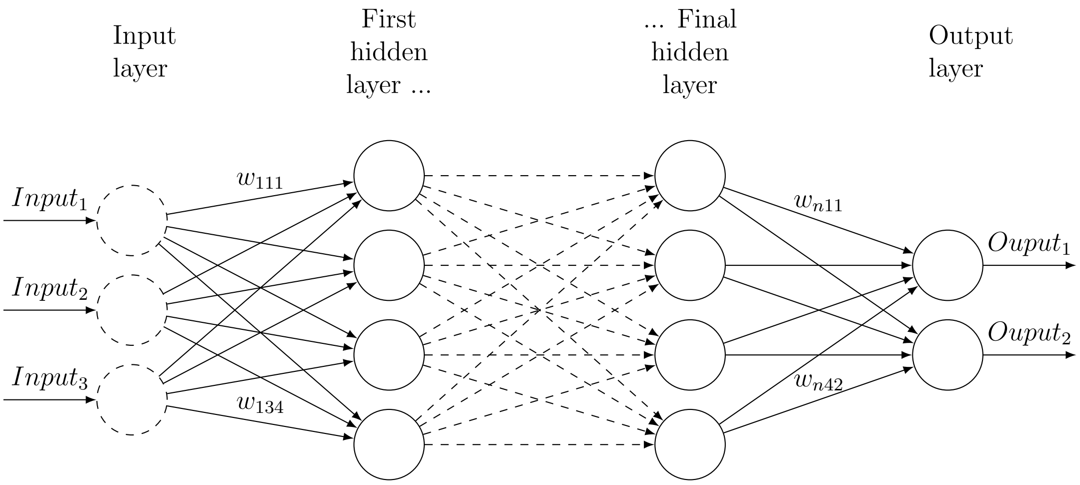

Figure 2.2: Neural network

Each neuron consists of an activation function which takes the weighted sum of the previous layer's outputs as input as well as a bias. Let us define the following:

- $n_1, ..., n_k$ the neurons of the layer i.
- $m_1, ..., m_l$ the neurons of layer $i + 1$.
- f the activation function of the neurons.
- w_{ij} the weight of the link between output of neuron n_i and neuron m_j.
- $Bias(n_i)$ the bias of neuron i.
- $Out(n_i)$ the output of neuron i.

Thus, we have:

$$Out(m_i) = f(\sum_{x=1}^{k} Out(n_i) * w_{xi} + Bias(m_i)) \tag{2.1}$$

This behaviour is shown on Figure 2.3. Multiple activation functions exist, such as Rectified Linear Units (ReLU) and hyperbolic tangents.

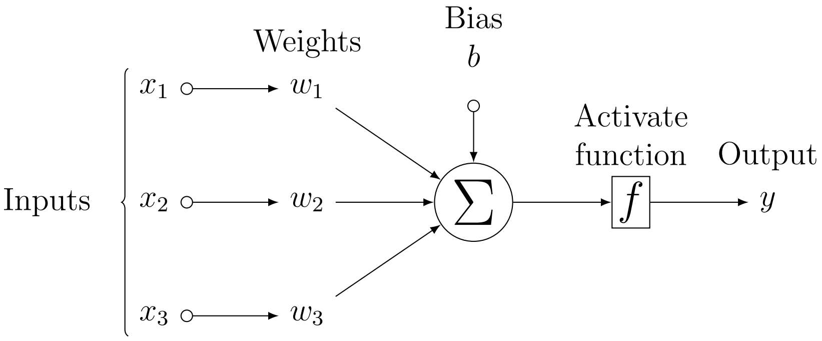

Figure 2.3: neuron definition

It is now quite easy to see how to get a prediction. One simply needs to feed an input to the network and propagate it thanks to Equation 2.1 in order to get the final output. This is called a feedforward propagation, since an input is passed from shallower layers to deeper ones. Now one needs to understand how to train the network. Let us denote by $\widehat{output}$ the output of the neural network computed with a feedforward pass and by $output$ the ground truth. We now define a loss function $l(output, \widehat{output})$. The mean square error and cross-entropy are often used as loss function respectively in regression and classification settings. The weights and biases of the networks are updated in order to minimise the loss function thanks to stochastic gradient descent. For example, to update the weight w_{ijk} one computes $\frac{d_l}{d_{w_{ijk}}}$ and makes the following update:

$$w_{ijk} \leftarrow w_{ijk} - \delta * \frac{d_l}{d_{w_{ijk}}}$$

with $\delta \in]0, 1]$. Note that those derivatives can be computed efficiently using back-propagation. Indeed, due to neural networks architecture, once the $\frac{d_l}{d_{w_{ijk}}}$ have been computed, they can be used to get the $\frac{d_l}{d_{w_{i-1jk}}}$ without carrying too much additional computations. This behaviour is shown in Subsubsection 3.2.2.3 with one of our formula which uses a similar technique to be computationally effective. Thus in order to make a training iteration, one has to make a feedforward pass through the network in order to compute the output of each neuron and get $\widehat{output}$. Once done, each $\frac{d_l}{d_{w_{ijk}}}$ are computed layer by layer (from deeper ones to shallower ones) while the network's weights are updated accordingly. In practice, the inputs are fed by batches of samples to the network. Therefore, a training iteration consists of computing all the weights update (one per sample) and adding them all together before updating the network's parameters. Batches are usually taken at random from the dataset (while making sure that each data sample appears equally) such that the network sees different batches most of the time. Finally, note that an epoch represents the number of training iterations to be done such that

$$nbr\ training\ iterations * batch\ size = nbr\ training\ samples$$

In other words, an epoch represents the number of training iterations required for the network to have seen each training sample at least once. Usually, artificial neural networks have to be trained on many epochs in order to get good results.

2.3.2 Feature selection with neural networks

Neural network can be built in a plentitude of ways and are subject to many parameters, neural architecture being the first one. Indeed, neural networks can take many forms, ranging from very shallow to very deep and very narrow to very wide. Many constraints can also be added in the architecture itself, convolutional and encoder layers are some of them. All of these parameters can be changed regarding the problem we are facing. In our case we decided to **limit ourselves to test our algorithms on networks with fully connected hidden layers**. We did this choice since our data didn't give us a priori reasons to introduce structure into our network. Furthermore, this is the more generic and "simplest" architecture that can be found. However, all the algorithms that will be proposed in Chapter 3 could be used to handle neural architecture constraints such as convolution.

Another important parameter is the activation function, it was considered for a long time that the best activation functions were either sigmoids or hyperbolic tangent. However, it was recently shown empirically that Rectified Linear Units (ReLU) functions were very effective [Nair and Hinton, 2010] (note that $ReLU(x) = max(0, x)$). Since then, pretty much **all neural networks have been built with ReLU neurons**, which has proven to be extremely effective. We thus decided to carry out our tests with such neural networks.

"Dropout" is another often used technique. It has been proved to significantly reduce overfitting and is now used in most neural networks. This technique was proposed in [Srivastava et al., 2014] and will be implemented in all our neural networks. Dropout can be seen as an "ensemble" method for neural networks. Indeed, the principle is to train only a subpart of the network at each iteration. In fact, each neuron has a given probability to be temporarily "removed" at train time. At test time, all neurons are used and their weights are adapted regarding their probability of being kept at training time. This can be seen as training multiple networks and averaging their predictions at test time (although this is not really what happens, but it would be too costly to train multiple networks). **By using dropout in our networks** we show that our algorithms are able to carry out feature selection even though each training iteration only trains a subset of the neural network.

Finally, we have to choose which training algorithm to use. All of them are based on the gradient descent principle and use back-propagation ([Rumelhart et al., 1988]) in order to be computationally efficient. ADAM is a recent algorithm that has been proposed in [Kingma and Ba, 2014] and which revolves around modifying the gradient descent in order to include momentum. The momentum introduced implies that the gradient descent has a sort of "short memory", i.e. while optimising the weights on a given data batch it also uses the derivatives computed on some of the previous batches. Due to its good performances, we decided to **train all of our networks with the ADAM optimiser**. The loss function used was either the accuracy in classification settings either the mean square error in regression settings. It is also worth noting that we normalise all our data such that each input and output have a 0 mean and 1 standard deviation, as it is often advised.

2.3.3 Previously tried feature selection methods

Even though there is very little literature regarding feature selection with neural networks, it has nonetheless already been looked at. We will now detail some of the methods that have already been proposed, most of which have been introduced very recently.

- In 1999, [Leray and Gallinari, 1999] proposed multiple formulae in order to carry out feature selection. They were separated into three categories: zero order, first order and second order methods. Zero order formulae were directly based on the parameters of the network while first and second order methods were respectively based on the derivative and second derivatives of those parameters. As an example of zero-order formula we have:

$$S_i = \sum_{j \in H} \frac{|w_{ij}|}{\sum_{i' \in I} |w_{i'j}|} * \sum_{k \in O} \frac{|w_{jk}|}{\sum_{j' \in H} |w_{j'k}|} \tag{2.2}$$

  where I, H, O denote respectively the input, hidden and output layer and S_i the importance of feature i. However, all of the formulae presented in that paper share the same drawback of only working for extremely shallow networks. In fact, this is highlighted in equation 2.2 where it appears that the network is assumed to consist of a single hidden layer. Furthermore, the tests in that paper were all done using neural networks with one hidden layer of 10 neurons, which is way too small for today's standards.

- In 2016, [Kim et al., 2016] proposed to use one of the formulae mentioned in [Leray and Gallinari, 1999] to tackle deeper neural networks. Indeed, in [Kim et al., 2016] a back-propagation method is used to compute feature importances. Let i be the neuron whose importance score we are calculating, and N_i the set of neurons in the next layer (closer to output) that i feeds into. The feature importance S_i is then computed as follow:

$$S_i = \sum_{j \in N_i} |w_{ij}| * S_j$$

  The back-propagation pattern clearly appears here since we need to know the importances of the deeper neurons in order to compute the shallower ones. Note that the formula has to be initialised and the output neuron(s) need to be given an initial importance. In [Kim et al., 2016], they only tackle a single output problem and address the initialisation by setting the importance of the output neuron to 1. However, this is a limitation and the multi-output setting is not discussed.

- Another recent article [Debaditya et al., 2015] gave some insight on how to associate neural activation and feature importances. The idea here consists of analysing the activation of the neurons for each input sample and averaging over all samples, thus using each data sample values and not basing the formula only on the network's intern parameters. This technique is proposed in [Debaditya et al., 2015] and works as follows. Let x_i be the i^{th} dimension of the input example x connected to j^{th} hidden neuron by w_{ji} and b_j the bias of hidden neuron j, the activation potential is calculated as

  $$a_{ij} = w_{ji} * x_i + b_j$$

  Then the average absolute activation potential contributed by the i^{th} dimension of M training examples $x^{(1)}, x^{(2)}, ..., x^{(k)} ..., x^{(M)}$ connected to j^{th} hidden neuron is given by:

  $$p_{ij} = \frac{1}{M} \sum_{k=1}^{M} |a_{ij}^{(k)}|$$

  where k represents k^{th} training example $x^{(k)}$. Afterwards, they compute the relative contribution of the i^{th} input dimension towards the activation potential of j^{th} hidden neuron using the following formula:

  $$c_{ij} = \frac{a_{ij}}{\sum_{i=1}^{N_{inp}} p_{ij}}$$

  where N_{inp} is the dimension of input example x. The net positive contribution c_i^+ of an input dimension i over all hidden neurons is given as:

  $$c_i^+ = \sum_{i=1}^{N_{hid}} ReLU(c_{ij})$$

  They use the ReLU function since negative c_{ij} can be set to 0 to show that i^{th} input dimension does not contribute to the activation of j^{th} hidden neuron. They finally state that the c_i^+ is a good measure to evaluate the feature importance. However, **they only look at the first layer of the network with this formula**.

- The last method we will mention has been proposed in [Li et al., 2015]. Their idea is to introduce a one-to-one connected layer as the first hidden layer of the network. The weights of the network are then regularised using elastic-net penalty [Zou and Hastie, 2005]. The absolute values of the one-to-one weights are then considered as being the feature importances.

2.3.4 Motivation and introduction to our proposed methods

For all the methods mentioned above, the tests made were very precise. The results shown were obtained on very particular datasets with extremely precise architectures. Furthermore, some of these techniques were only tried with very shallow neural networks (i.e. all of the formulae from [Leray and Gallinari, 1999]). We have been inspired thanks to the literature we just reviewed to create or modify some of those feature selection methods. Hereunder is a list of the algorithms we will propose in Chapter 3, each of which is linked to the literature it comes from:

- We will first look at the method proposed in [Li et al., 2015] and will analyse the effect of regularisation.
- We will then propose a brand new algorithm which is a mix between the methods proposed in [Debaditya et al., 2015] and [Kim et al., 2016]. Indeed, the first article ([Debaditya et al., 2015]) defines a measure of the activation potential of each neuron by averaging over all input samples whereas the second one ([Kim et al., 2016]) uses a back-propagation technique in order to transmit neuron importance from deeper layers to shallower ones. We propose a method that combines both by creating a new zero-order formula which uses back-propagation as well as neural activation, thus having to be averaged over all samples. Note that two very recent articles ([Montavon et al., 2017] and [Shrikumar et al., 2017]) have been published (in May and April 2017 respectively) and propose very similar techniques. In fact, one of the formulae proposed in [Montavon et al., 2017] ends up being almost the exact same as a formula that we propose ourselves, which gives another theoretical motivation for our method.
- We will then show that the "one-to-one layer" technique ([Li et al., 2015]) can be added to our back-propagation in order to boost performances and add a regularisation term easily.
- We will finally study a feature swapping technique as explained in [Qi, 2012]. Note that while our other proposed algorithms are embedded, this one is a wrapper.

We didn't want to limit ourselves to analysing a specific kind of problem and wanted to show that our algorithms are general enough to work on multiple datasets, we thus tested them on artificial classification and regression datasets. Furthermore, we will analyse the effect of the architecture of the networks and show that they remain stable for deeper networks. Finally, we will give a comparison of all our proposed algorithms and mention their pros and cons.

Overall, we will show that by combining all of the previously proposed techniques we are able to achieve reliable feature selection with deeper neural networks resulting in good performances. We will also propose extensions to most of our algorithms, some of which might be of high interest for further studies. By doing this we hope to fill in a gap that exists in the literature about feature selection with deep neural networks.

Chapter 3

Methods development and explanations

In this chapter, we will introduce five different methods to implement feature selection with neural networks. Section 3.1 proposes to modify neural architecture by adding a one-to-one connected layer at the beginning of the network, as proposed in [Li et al., 2015]. Extensions to that technique will also be discussed in Subsection 3.1.5. Section 3.2 gives an algorithm to compute feature importance using back propagation techniques based on two formulae. In particular, Subsection 3.2.1 presents an algorithm using a formula based on the outputs of each neuron (of order zero [Leray and Gallinari, 1999]) while the algorithm presented in Subsection 3.2.2 is based on their derivatives (of order one [Leray and Gallinari, 1999]). Two important extensions will be discussed in Subsection 3.2.1, we will see that this technique allows for a better understanding of the network and could lead to network pruning without accuracy loss. Section 3.3 will show that it is easy to mix the techniques presented in Section 3.1 and Section 3.2.1 to enhance the results. The last technique is introduced in Section 3.4 and consists in randomising inputs and measuring performances loss. In Section 3.5 we will give pros and cons of each algorithm and compare their performances on different artificial datasets, one of which is taken from [Li et al., 2015].

3.1 Invisible layer (I.L.)

The technique we will present in this section aims at penalising the use of too many variables, forcing the network to use (and thus to find) the interesting features among all others.

3.1.1 Graph conventions

Let us introduce the graph conventions that will be used throughout this Section. First, it is important to note that two different types of neurons will be used, so we should be able to distinguish them. Both types work the same way but differ in their activation function.

- Type 1 (which will be referred to as “ReLU” neuron) uses the ReLU activation function, and will be represented as a circle. Also note that a bias b is added to these neurons. We thus have :

$$y = \sum_{i=0}^{n} max(0, x_i * w_i + b)$$

- Type 2 (which will be referred to as “identity” neuron) uses the identity as activation function and will be represented as a dashed circle. This leads to :

$$y = \sum_{i=0}^{n} x_i * w_i$$

Type 2 neurons can be considered as “invisible” for the network, and will only be used to build the one-to-one connected layer, giving its name to the “invisible layer” technique. A graphical definition of each neuron and their representation is given in Figure 3.1. There is another convention that will be used throughout this document. Whenever two neurons are connected with a dashed line, it means that there are not in fact directly connected and that some hidden layers have not been drawn for ease of understanding. An example of this can be seen on Figure 3.2.

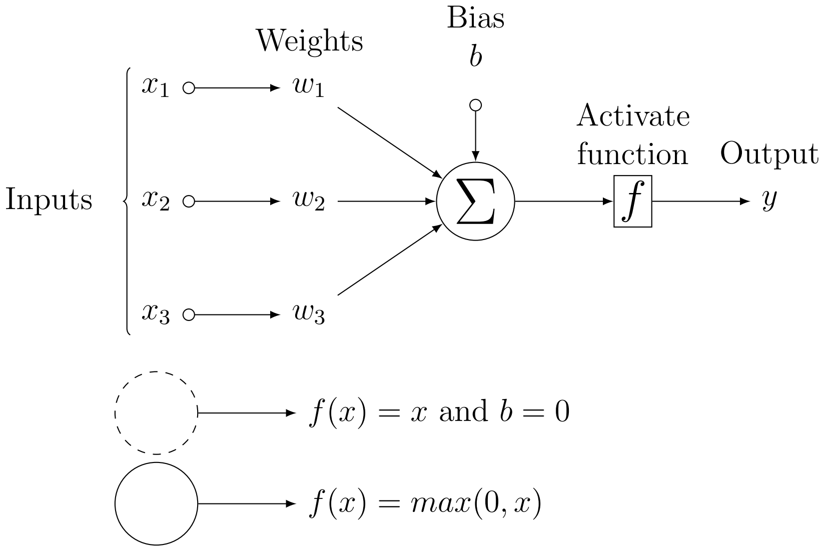

Figure 3.1: neurons definition

3.1.2 Method principle

We will now detail the invisible layer technique. In order to use this method, we first need to introduce a one-to-one layer at the beginning of the network, note that this layer has to be made of “identity” neurons. Indeed, with “ReLU” neurons:

- Let i_n be the weight linking $input_n$ to neuron n of the invisible layer (as shown on Figure 3.2).

- Let o_n be the output of neuron n of the invisible layer.

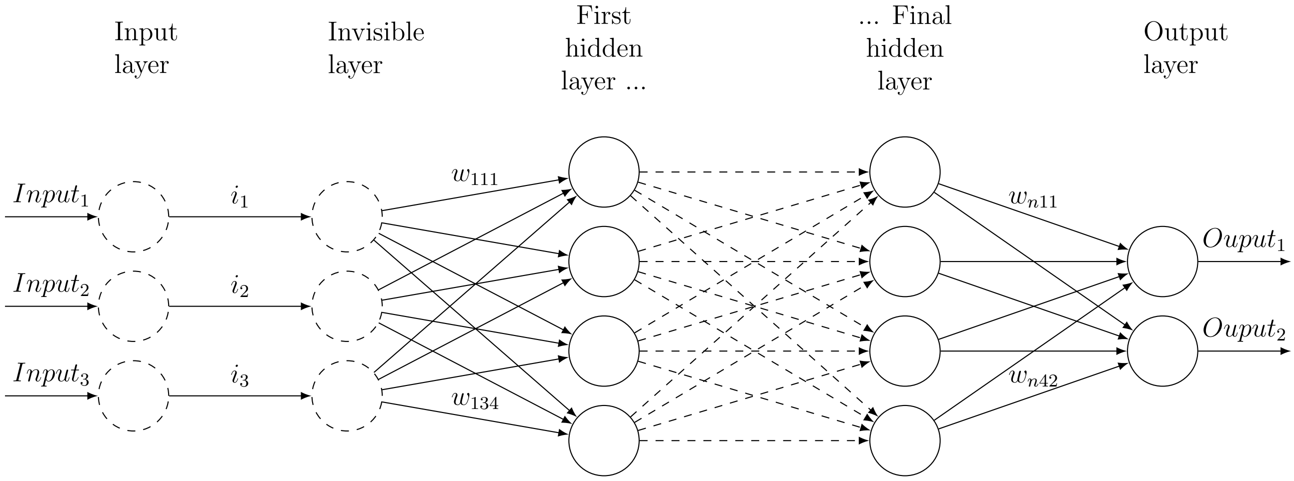

Figure 3.2: Example of invisible layer architecture

Then we have :

$$o_n = \begin{cases} 0, & \text{if } i_n * input_n \leq 0 \\ i_n * input_n, & \text{if } i_n * input_n > 0 \end{cases} \tag{3.1}$$

Of course, this is unacceptable since whether a feature has negative or positive values it needs to be passed to the network. However, as we can see in Equation 3.1 using ReLU neurons would only allow either positive or negative values to pass through the invisible layer. This is why “identity” neurons are used in this part of the network. With the layer in place, it is in fact very easy to get the feature importances. The idea is that the network will adapt the invisible layer weights in order to promote informative variables over others. Since this is done by increasing the absolute value of the weights, it is sufficient to compare them, leading to (according to Figure 3.2):

$$importance(input_n) = |i_n|$$

Remember that this comparison can be made as such since the inputs have been normalised (each of the inputs has a mean of 0 and a standard deviation of 1). If it wasn't the case, inputs values should be taken into account such that none are favoured over others.

3.1.3 Regularisation

Even though the network should adapt its weights naturally, it is a good idea to investigate how we could make the weights difference between informative and non-informative variable even greater. For this purpose, an elaborate regularisation technique based on the Lasso and elastic-net methods has been developed in [Li et al., 2015]. We will use a simpler similar technique, which penalises the network for leaving high weights in the invisible layer. This way, the network should be more inclined to bring the weights of useless features down. To this end, we modify the error term that has to be minimised while training. Indeed we have (with the same notations as in Figure 3.2):

$$cost = f(output_1, ..., output_j; real\_outputs) + \alpha * \frac{\sum_{k=1}^{n} |i_k|}{n} \tag{3.2}$$

where f is either the cross-entropy or the mean-square error function (depending on whether it is a classification or regression problem respectively) and where the higher α, the bigger the regularisation. Initially all $i_n = 1$, such that inputs are passed without any modification to the network (other initialisations have been tried, but were found to give similar or worse results). In theory, this regularisation leads to the problem that a closer-to-optimum solution can always be found by diminishing all i_n and increasing deeper weights in the network. Even though we didn't find it problematic in practice, we tried to counter that by trying another regularisation technique which led to the following cost function:

$$cost = f(output_1, ..., output_j; real\_outputs) + \alpha * \frac{\sum_{k=1}^{n} |i_k|}{n} + \alpha_2 * \frac{\sum_{w \in W} |w|}{||W||}$$

where W represents the ensemble of the weights not belonging to the invisible layer and $||W||$ the number of weights in W. This formula is known to converge only to a local minimum and often leads to instability with deeper networks (as mentioned in [Li et al., 2015]). This effect can be seen on Figure 3.4, where it appears that due to α_2 being too big, the network is unable to learn correctly. Thus we decided to keep using Equation 3.2 by leaving $\alpha_2 = 0$.

Also note that we computed the mean value of the invisible layer's weights rather than the sum. This allows us to have for the first training iteration (regardless the size of the invisible layer):

$$\frac{\sum_{k=1}^{n} |i_k|}{n} = 1$$

This is important since we want to know precisely the ratio between the regularisation term and the error term to tune α. Therefore, for training step one we have:

$$\frac{\sum_{i=k}^{n} |i_k|}{n * f(output_1, ..., output_n; real\_outputs)} = \frac{1}{f(output_1, ..., output_n; real\_outputs)}$$

Since the data is normalised, we know approximately the error that will be made $\hat{f}$ (as if it was random guessing), and can define an α such that:

$$\alpha = p * \hat{f}(output_1, ..., output_n; real\_outputs)$$

where we have that p represents the importance of the regularisation term, compared to the error term. For example, if $p = 1$, both terms will more or less have the same value at training step one. This has to be the case since a big regularisation term would become too significant compared to the error term, making the network unable to learn. This effect can be seen on Figure 3.3

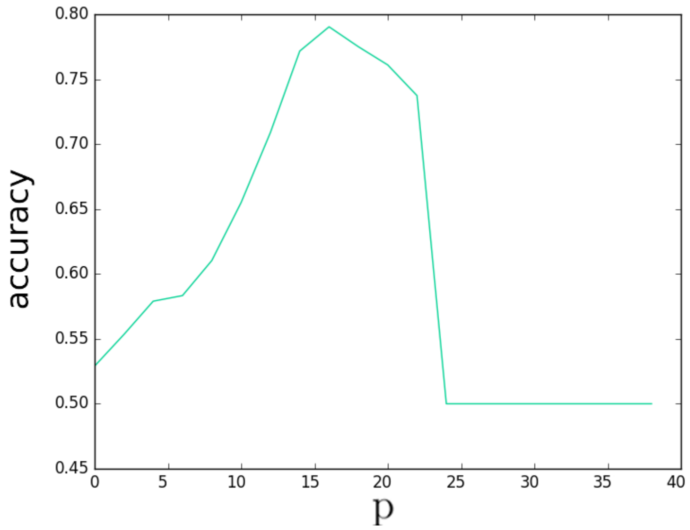

Figure 3.3: Effect of the parameter α (through p) on the accuracy of the model. The results are obtained with 15000 training steps, and with a network of 4 hidden layers of 500 neurons each. These results were obtained on a noised hypercube binary classification problem generated by Scikit-learn [skl, b](50 informative input features for 4950 non-informative input features). In this case, 2000 training samples were used.

3.1.4 Results obtained

We trained networks with α equal to $[0, 1, 6, 6]$ and α_2 equal to $[0, 0, 0, 2]$ respectively on a binary classification problem. Each of the networks is composed of 4 hidden layers of 300 neurons each. The dataset was generated using an hypercube of dimension 25 where each vertex is mapped to one of two classes. Furthermore, we added 4975 irrelevant features generated with different noise types such that each networks takes 5000 inputs. In this case we generated 3000 training samples and computed the accuracy on 5000 other samples. More information on the dataset generation is given in Appendix A. The results can be seen on Figure 3.4 which clearly shows the effect of regularisation. As we can see on the top graph, regularisation greatly improves the learning curve as well as the accuracy of the network. We can also see that in terms of feature selection performance, increasing the regularisation greatly helps improving AUPR score (middle graph). The bottom graph shows the evolution of the i_n, where the dashed line represents the mean weights corresponding to useless features, and where the continuous line represents the mean weights corresponding to useful features. As expected, as regularisation increases, weights tend to become smaller. We can also see that regularisation increases the proportional gap between the dashed and the non-dashed lines, which is wanted. However we see that with a too high regularisation ($\alpha_2 = 2.0$) the focus is on decreasing the invisible layer's weights rather than minimising the error, thus destabilising the network. In that particular case, the accuracy is not any better than random guessing, and the AUPR is abysmal.

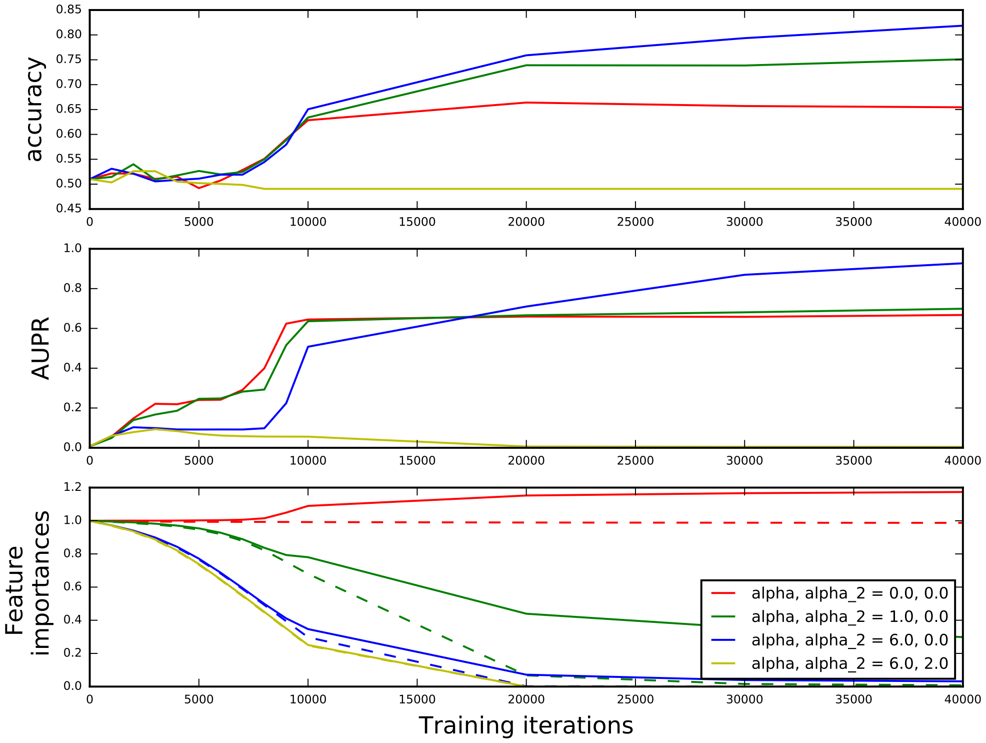

Figure 3.4: Accuracy, AUPR and weights evolution of a network with different regularisation parameters. Each line color corresponds to a different α, α_2 combination. In the bottom graph, the filled lines represent the average importance of the relevant inputs whereas dashed lines represent the average importance of the non-relevant inputs.

3.1.5 Extensions

Unfortunately the regularisation method we used doesn't allow to select redundant features. Indeed, imagine we have two features representing the same information. Since the regularisation is linear, it is equivalent cost-wise to have one big and one small weights rather than two medium one (note that this is beneficial when one wants to select as few variables as possible, i.e. to solve the *minimal optimal problem* as explained in Section 2.1). To counter this (i.e. to solve the *all-relevant problem*), quadratic regularisation (**elastic net**) could be introduced and would help the network selecting both of the variables in order to minimise the cost.

Another extension would be the ability to introduce a priori knowledge. This could probably be done by using the following formula :

$$cost = f(output_1, ..., output_n; real\_outputs) + \alpha * \frac{\sum_{i=k}^{n} |i_k| * \frac{1}{w_k}}{n} \tag{3.3}$$

where w_k is proportional to the a priori importance of a feature. This way, if we know a priori that some features are less informative, we can penalise them more than others in the regularisation term.

A final extension would be to find a way to introduce a "structured invisible layer" (i.e. more complex than one-to-one, similarly to convolution), such that we could regularise groups of inputs, if we know a priori that these groups of features act together.

3.2 Back-propagation techniques (B.P.)

In this section, we propose two different methods which do not require a modified architecture. Both use back-propagation (a process that is used for training neural networks [Rumelhart et al., 1988]) in order to compute neuron importances. As mentioned in Chapter 2, a back-propagation method has already been proposed in [Kim et al., 2016]. However, only intern parameters are used with that method, whereas ours also uses data samples to compute neural activation (similarly to [Debaditya et al., 2015]). Our algorithm is presented hereunder:

Algorithm III. *General algorithm for back-propagation feature selection methods.*

1. Train a network (or use a pre-trained network).
2. For each training sample, do the following steps :
   (a) **Initialisation phase:** Assign importances to the neurons of the network's last layer, by propagating the training sample through the network.
   (b) **Back-propagation (step one):** Use the importances of neurons from layer i to compute those of layer $i-1$, where layer i is the one for which importances have already been assigned and is the furthest away from the output.
   (c) **Back-propagation (step two):** Repeat step (b) until importances have been assigned to the input layer's neurons.
   (d) **Store importances:** The features importances for this input sample correspond to the neurons importance of the input layer and need to be stored.
3. Repeat step (2) for each training sample and sum all the feature importances.
4. The sum finally obtained corresponds to the overall feature importances.

It appears here that some additional computations have to be carried out to get the feature importances. Indeed, we need to run the algorithm after training the network, whereas for the invisible layer method fetching the weights was enough. Note that we can give an estimate of the computation time. Indeed, the algorithm only needs to be run on the training samples. Therefore, this leads to a **computation time of the order of a training epoch**. Subsection 3.2.1 and Subsection 3.2.2 aim at giving two different ways of computing the importances based on the principle described above.

3.2.1 Importance backpropagation

The goal of this subsection is to give a formula to compute an importance defined as how much a given neuron contributes to the output "variability", to this end we will go through the 5 following parts :

1. **Importance metric definition.** This paragraph will define an importance measure of each neuron for a given data sample. This measure is based on neural activation. As mentioned in Chapter 2, such technique has already been proposed in [Debaditya et al., 2015]. However, they only analyse the contribution of the inputs on the first hidden layer, whereas here we propose a formula that takes the whole network structure into account.
2. **Initialisation.** In this paragraph, a method for initialising the algorithm will be discussed. We will also give some clues on how this technique could be refined in different settings.
3. **Backpropagation.** We will explain how the two first parts are put together to obtain the algorithm.
4. **Results.** We will show an example of importances that are obtained using this method and show that the results seem reasonable.
5. **Extensions.** In this subsubsection we will show that given the results obtained, this method might also be used in order to prune neural networks without hurting accuracy.

3.2.1.1 Importance metric definition.

We will start by explaining the back propagation formula that is used in order to compute the neurons importances. In order to achieve this, we will suppose that we know a priori the last hidden layer's importances. We will discuss the initialisation problem afterwards (3.2.1.2). Say we want to compute the importances of a given layer, let us define for a data sample (as can be seen on Figure 3.5):

- $n_1, ..., n_k$ the neurons of the layer studied.
- $m_1, ..., m_l$ the neurons of the next layer (for which we already know the importances).

- f the activation function of the neurons.
- w_{ij} the weight of the link between output of neuron n_i and neuron m_j.
- $Bias(n_i)$ the bias of neuron i.
- $Out(n_i)$ the output of neuron i. Note that we have :

$$Out(m_i) = f(\sum_{x=1}^{k} Out(n_i) * w_{xi} + Bias(m_i))$$

- $Imp(n_i)$ the importance of neuron i.

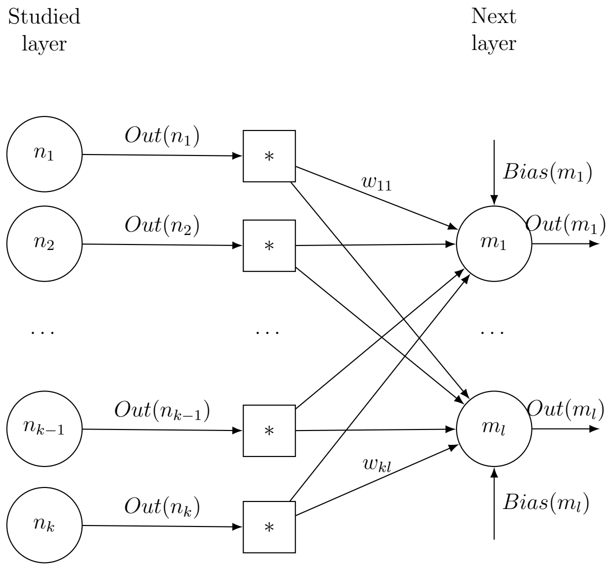

Figure 3.5: Graphical representation of network definitions

This allows us to define the following equations:

$$Total_x = \sum_{i=1}^{k} f(Out(n_i) * w_{ix}) \tag{3.4}$$

$$Imp(n_i) = \sum_{x=1}^{l} \frac{f(Out(n_i) * w_{ix}) * Imp(m_x)}{Total_x} \tag{3.5}$$

We will now detail the idea behind equation 3.5. The goal is to measure the role that each neuron plays on the next layer. This also has to be balanced according to the role of the next layer's neurons. Let us analyse Equation 3.5 term by term.

- First we have $f(Out(n_i) * w_{ix})$ which represents the contribution of one particular neuron i on one particular neuron j of the next layer.
- Then we have, $Total_x$ (defined by equation 3.4) represents the overall contribution of all neurons from the layer studied on a neuron x of the following layer. We decided to take the activation function of each contribution in order to take only "positive impacts" into account (as is also suggested in [Debaditya et al., 2015]). This is especially true since the activation function used are ReLU and are therefore linear (if $x >= 0$).
- Since each neuron is not used in the same way, their usefulness varies. Therefore we need to introduce a term to take these differences into account. This way, if a neuron has a big contribution on another useless neuron, it will be considered as being less useful than a neuron which has a medium contribution on a very useful neuron. Multiplying by $Imp(m_x)$ achieves this goal. Note that this only holds if we use ReLU activation functions. In fact, we tried this formula with hyperbolic tangent activation functions and obtained poor results. This implies that the formula should probably be modified to work with other activation functions.

All in all we have, $\frac{f(Out(n_i)*w_{ix})}{Total_x}$ which represents the relative contribution of a particular neuron compared to others ($\in n_i$) on another neuron ($\in m_i$). This is then multiplied by $Imp(m_x)$ in order to scale that contribution according to the usefulness of the following layer's neurons. Finally we sum all of this for each m_i, giving us an idea of the global impact of each n_i on the next layer.

3.2.1.2 Initialisation.

Now that the back propagation step has been clearly defined, we need to tackle the initialisation problem. Indeed, this formula works quite well but needs the a priori knowledge of importances for it to work. Let us start by explaining how we initialise the importances for regression problems, we will then expand on the method used for classification problems.

In a single output regression problem setting, we also need to consider negative output values the same way as positive ones. This leads to the following initialisation (with w_i the weights connecting the last hidden layer to the output) :

$$Imp(n_i) = |Out(n_i) * w_i| \tag{3.6}$$

In multi-output regression settings, we make the hypothesis that each output neuron has the same importance. This way, if we let $n_1, ..., n_k$ be the neurons of the last hidden layer and $m_1, ..., m_l$ be the output neurons ($l = 1$ if it is a single output problem). We have (with w_{ix} the weights connecting the last hidden layer to output x):

$$Imp(n_i) = \sum_{x=1}^{l} |Out(n_i) * w_{ix}| \tag{3.7}$$

Equation 3.7 represents the fact that the importance of a neuron from the last hidden layer in a regression case corresponds to the sum of all its contributions on the predictions. Where a negative contribution is considered as important as a positive contribution. Note that this only works if all the outputs are normalised such that they have a mean equal to 0 and a standard deviation equal to 1. If this wasn't the case, we should consider different importances for the output neurons, depending on their mean value (the higher, the less important) such that in the end, all output variables are considered as equally important. Finally, notice that in a multi-regression problem it is very easy to obtain feature importances for each output separately, we only need to run the algorithm by initialising output importances to 0 except the outputs for which we want to select features. This technique will in fact be discussed in Chapter 4 in the context of multi-output gene regression, where we are interested by the information contained in the inputs gene by gene.

In classification problems, softmax layers are often added before the output layer such that the output neurons correspond to probabilities of belonging to a given class. Softmax layers map a N-dimensional vector v of arbitrary real values to another N-dimensional vector $soft(v)$ with values in the range $[0, ..., 1]$ that add up to 1. This is done by using the following formula:

$$soft(v)_j = \frac{e^{v_j}}{\sum_{n=1}^{N} e^{v_n}}, \text{ for j = 1,...,n}$$

In our case, we use the same back-propagation formula for classification problems, but apply it by considering the inputs of the softmax layer as being the output (since the output itself consists of probabilities). This method might not be optimal since we consider each neuron of the softmax layer as equally important, and we do not consider negative and positive values differently. For example, for a binary classification problem, we make no difference if the inputs of the softmax layer are $[0.7, 0.2]$ or $[0.7, -0.2]$. With this example, the differences are equal to 0.5 and 0.9 respectively. This would lead to two way different output probabilities after the softmax layer.

Note that we tried to consider negative importances in this case (for example a negative contribution on a softmax neuron would lead to a negative importance). This led to better results part of the time, but most often than not destabilised the formula. Another idea would be to introduce a measure of difference between the neurons of the softmax layer, which would be easy to do in a binary classification case. Basically the more the output of a neuron increases the difference between the two softmax neurons, the higher its importance. For example, if we have a neuron n that is connected with w_1 to softmax neuron 1 and with w_2 to softmax neuron 2, we could initialise the importance with the following formula:

$$Imp(n) = |w_1 Out(n) - w_2 Out(n)| \tag{3.8}$$

Using this formula would be much better in the case of binary classification since this is exactly what we want to compute with feature selection. To introduce a similar concept with more than two classes, some measure on the entropy could be used. Since the formula works quite well already by using the same technique than for regression, we will stick with it through this report such that we talk about a formula with applications as broad as possible. Keep in mind that it might not be the best initialisation possible for classification problems. In fact, the same goes for regression if the outputs of the problem are not supposed to be of equal importance (or are not normalised).

3.2.1.3 Backpropagation.

Once the initialisation has been done we can use the formula defined by equation 3.5 in order to get a measure of the importance of each neuron for a given data sample. To do this, we make a forward pass by feeding the input to the network and computing the output of each neuron. Once this phase is done, we initialise the importances of the last hidden layer as explained before. Finally we repeat equation 3.5 until we get the input neurons importances. We finally sum this importance for all of the data samples in order to have the general importance of each neuron.

3.2.1.4 Results.

We are now going to look at the results for a regression problem. The dataset we will use has 5000 input features $[x_1, ..., x_{5000}]$. The regression problem has been generated using the following formula, where the weights have been chosen uniformly at random between 0 and 100 (the dataset has been generated using scikit-learn [skl, c]):

$$y = \sum_{i=1}^{25} w_i * x_i$$

This means that only 25 out of the 5000 input features are actually useful to predict the output. Figure 3.6 shows the importance of each neuron (layer by layer). In this case the neural network used has 4 hidden layers of 500 neurons each (x axis of Figure 3.6). Layer 0 is the input layer and thus corresponds to the feature importances. As we can see, there are peak importances on the first neurons of layer 0, this is expected (due to the problem nature, which only uses the 25 first features) and proves that the technique used seems reasonable.

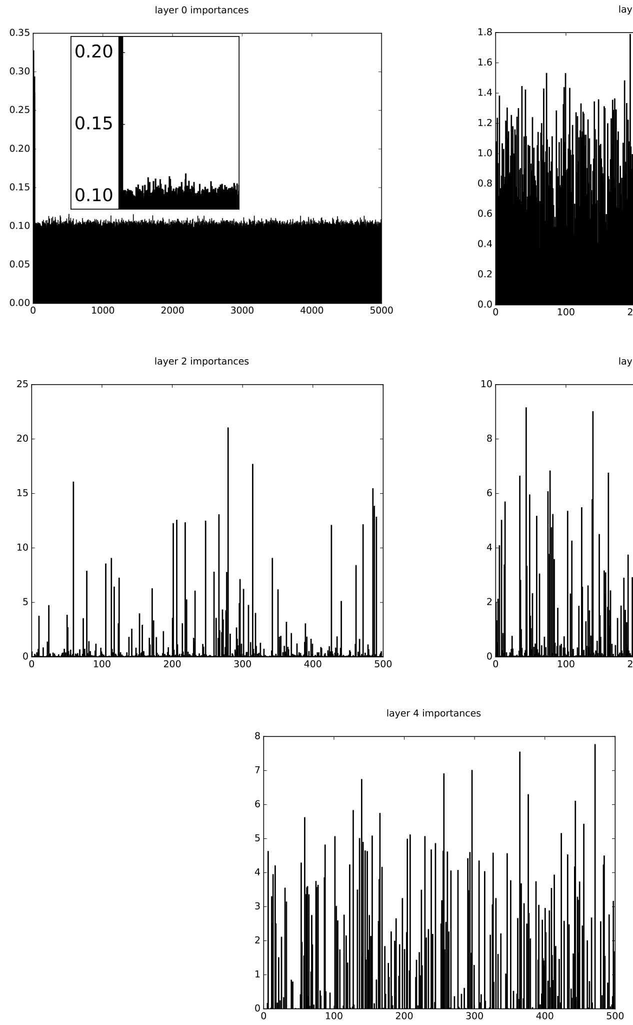

Figure 3.6: Neuron importances computed using equations 3.4 and 3.5 where $f = ReLU(x)$

3.2.1.5 Extension.

There is a very interesting thing that appears when looking at Figure 3.6. Indeed, some neurons seem to have very low importances. For the last layer it can only mean that some of the neurons are actually never activated. For the hidden layers, it either means that neurons are never activated or that all the weights exiting them are negative (or positive but contribute useless neurons). Therefore we decided to analyse this behaviour a bit deeper by letting $f = abs()$ in equations 3.4 and 3.5, leading to negative weights being considered equally as positive. If we find 0 importance neurons using this formula, it means that they are in fact never active and can be pruned. The formulae used to compute feature importances in this case were thus:

$$Total_i = \sum_{x=1}^{k} |Out(n_x) * w_{xi}| \tag{3.9}$$

$$Imp(n_i) = \sum_{x=1}^{l} \frac{|Out(n_i) * w_{ix}| * Imp(m_x)}{Total_x} \tag{3.10}$$

This method was worse in terms of feature selection as can be seen on Figure 3.7, but using it allowed us to prove that there were still 0 importance neurons (which can only be due to no activation in this case). To prove our sayings, Figure 3.8 shows the neurons importances with $f = abs()$ used in equations 3.4 and 3.5 instead of $f = ReLU()$, the network and dataset used are exactly the same as for Figure 3.6.

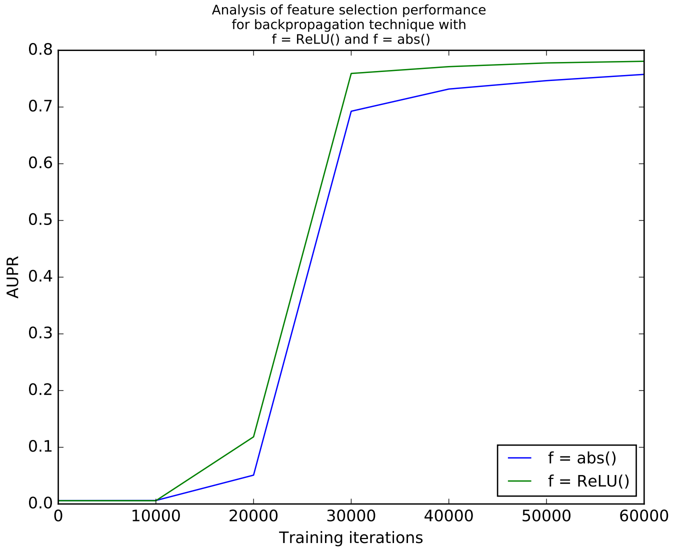

Figure 3.7: Comparison between of feature selection performance based on equations 3.4 – 3.5 and equations 3.9 – 3.10. The network and dataset used are the same as for Figures 3.6 and 3.8.

This means that we would be able to prune a good proportion of some layers by running the algorithm with $f = abs()$ and removing 0 importance neurons. Furthermore, it is possible that neurons with very little impact are in fact "noisy" neurons which favour overfitting, and removing them could probably increase accuracy, although these are only speculations and have not been studied in this context.

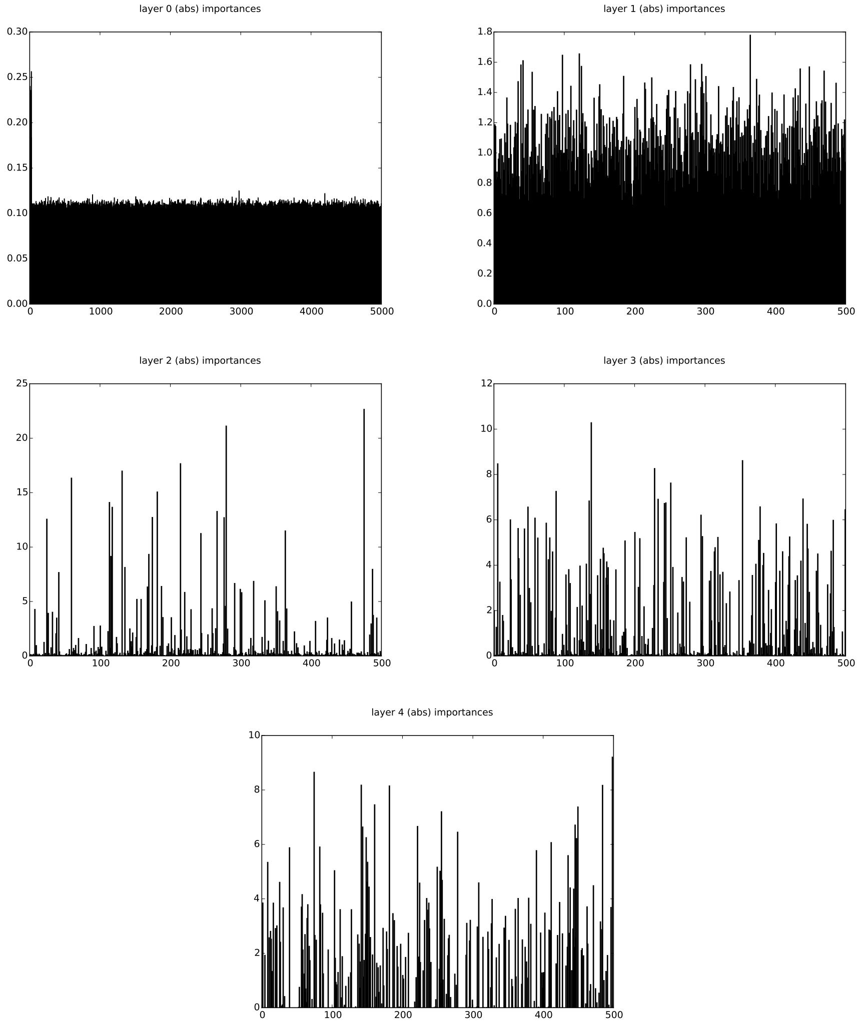

Figure 3.8: Neuron importances computed with equations 3.9 and 3.10

Another extension can be proposed. Indeed, the back-propagation method gives information sample by sample. The methodology described above simply summed up the importance over every data sample, but the algorithm could be used differently. For example, it would be possible to analyse group of features. This could be done by looking at "mutual importances". Imagine we run the algorithm on a given data sample and have :

$$\begin{cases} Imp(input_0; sample) = x_0 \\ Imp(input_1; sample) = x_1 \end{cases}$$

We could define the mutual importance as follows:

$$Imp(input_0, input_1; sample) = x_0 * x_1$$

Averaging this measure over all samples would in fact lead to a measure that shows which neurons are more likely to activate simultaneously and would allow to distinguish which inputs are generally used together. Furthermore, in a classification problem, we could imagine to look at feature importances for the samples of a single class such that we have an idea of what features are used to predict that given class against the rest. All in all, this technique would allow to have a much better understanding of the network itself, making it less of a blackbox model.

3.2.2 Derivative backpropagation

In this subsection, we give a new importance measure that aims at capturing the average sensitivity of outputs with respect to the input around the learning samples. We will focus on describing the importance metric and the initialisation phase as in Subsection 3.2.1. We will then discuss the results obtained.

3.2.2.1 Importance metric definition

We can now introduce the backpropagation formula based on derivation which seems like a logical choice. Indeed, the derivative of the network's output relative to a neuron output represents how much a little change in the input is repercuted on the output, it should therefore give a measure of importance. Methods based on derivation of neurons have already been discussed in [Leray and Gallinari, 1999], although they were discussed for small networks and few results were given. We will now detail our method. For simplicity, let us use the following shortcut in the writing of our equations:

$$\frac{dm_x}{dn_i} = \frac{dout(m_x)}{dout(n_i)}$$

Let us also introduce the following concepts :

- $f'()$ as the derivative function of $f()$.
- If $f(x) = ReLU(x) \rightarrow f'(x) = \frac{max(0,x)}{x} = \frac{f(x)}{x}$.
- $final_i = output_i$ which corresponds to one of the network's output. Note that in a classification problem, the outputs are taken at the entry of the softmax layer.

To compute $\frac{dfinal_k}{dinput_x}$ we will use the back propagation technique. This means that we find $\frac{dfinal_k}{dn_i}$ for each neuron, proceeding layer by layer. Thus by using the same notations as in Figure 3.5, we have:

$$\frac{dfinal_k}{dn_i} = \sum_{x=1}^{l} \frac{dfinal_k}{dm_x} * \frac{dm_x}{dn_i} = \sum_{x=1}^{l} \frac{dfinal_k}{dm_x} * w_{ix} * f'(\sum_{y=1}^{k} Out(n_y) * w_{yx} + Bias(m_x)) \tag{3.11}$$

Thus with o the number of outputs, $\sum_{k=1}^{o} |\frac{dfinal_k}{dn_i}|$ corresponds to the importance of neuron n_i. Note that we need to add absolute values since negative contributions are as significant as positive ones and that we make the assumption that a variation in each output has the same importance.

3.2.2.2 Initialisation

Using this technique no importances need to be initialised, although there is still a choice to be done. Indeed, the choice of the $final$ function is quite meaningful. In our case we chose to simply use the sum of every output, although this might not be the smartest choice. As stated in Subsection 3.2.1, this works well if output values are normalised and are supposed to have the same importance a priori. The same remark can be made regarding classification problems, where it might be better to consider “differences” as with Equation 3.7 in Subsection 3.2.1. Although in this case, the $final$ formula should remain differentiable.

3.2.2.3 Back-propagation and results

The back-propagation is also done by carrying out a forward pass before repeating equation 3.11 multiple times in order to get $\frac{dfinal_k}{dinputs_i}$ for a given sample. This is then summed over all data samples. Unfortunately the results obtained were quite bad, let us analyse the formula and explain why we think this was the case.

The activation function that was used throughout the network was the $ReLU$, leading to

$$f'(x) = \frac{max(0,x)}{x} = \frac{f(x)}{x}$$

This means that for one sample:

$$Imp(input_i) = \sum_{o=1}^{\text{nbr outputs}} (\sum_{\forall paths} (\prod_{w \in path} w)) \tag{3.12}$$

where :

- $\forall paths$ represents all non blocked paths going from $input_i$ to $output_o$. A path is said to be blocked if it goes through an inactive neuron, i.e. if the output of that neuron is 0.

To illustrate this behaviour, let us give a simple example, and carry out the derivation from start to finish. Let us use the network defined by Figure 3.9.

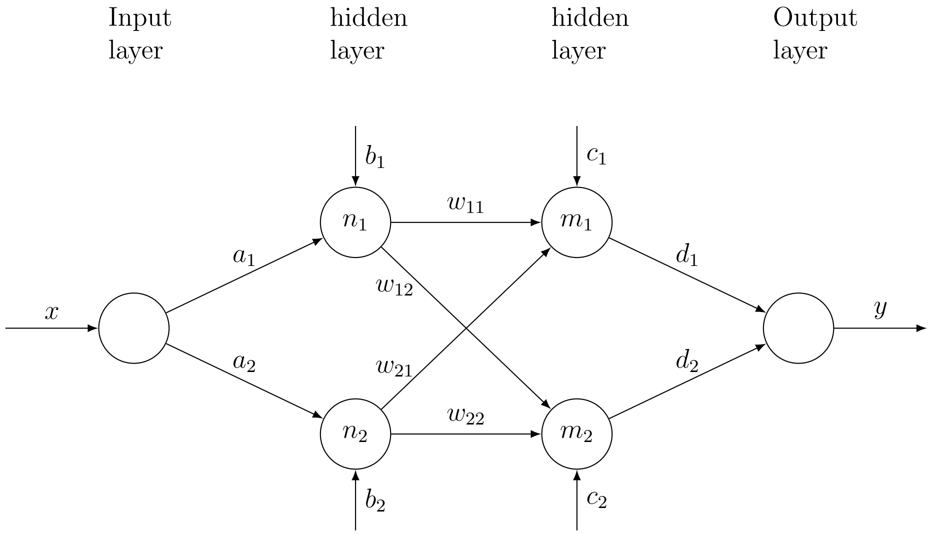

Figure 3.9: Network used to illustrate derivation

We have the following:

$$\begin{cases} y = d_1 out(m_1) + d_2 out(m_2) \\ out(m_1) = f(w_{11} out(n_1) + w_{21} out(n_2) + c_1) \\ out(m_2) = f(w_{12} out(n_1) + w_{22} out(n_2) + c_2) \\ out(n_1) = f(a_1 x + b_1) \\ out(n_2) = f(a_2 x + b_2) \end{cases}$$

Thus, we can easily derive the importances of neurons m_1 and m_2:

$$\begin{cases} \frac{dy}{dout(m_1)} = d_1 \\ \frac{dy}{dout(m_2)} = d_2 \end{cases}$$

Now, we develop y using the previous system and derive the importances of neurons n_1 and n_2:

$$\begin{cases} y = d_1 f(w_{11} out(n_1) + w_{12} out(n_2) + c_1) + d_2 f(w_{21} out(n_1) + w_{22} out(n_2) + c_2) \\ \frac{dy}{dout(n_i)} = d_1 w_{i1} f'(w_{11} out(n_1) + w_{12} out(n_2) + c_1) + d_2 w_{1i} f'(w_{21} out(n1) + w_{22} out(n2) + c_2) \end{cases}$$

Note that this could have been obtained more easily using the chain rule:

$$\begin{cases} \frac{dout(m_i)}{dout(n_j)} = w_{ji} f'(w_{1i} out(n_1) + w_{2i} out(n_2) + c_i) \\ \frac{dy}{dout(n_i)} = \frac{dy}{dout(m_1)} \frac{dout(m_1)}{dout(n_i)} + \frac{dy}{dout(m_2)} \frac{dout(m_2)}{dout(n_i)} \end{cases}$$

which leads to the same result and shows how back-propagation can be used to great effect for computation times. We now finally use the chain rule in order to get the input importance:

$$\begin{cases} \frac{dout(n_i)}{dx} = a_i f'(a_i x + b_i) \\ \frac{dy}{dx} = \frac{dy}{dout(n_1)} \frac{dout(n_1)}{dx} + \frac{dy}{dout(n_2)} \frac{dout(n_2)}{dx} \end{cases}$$

This finally leads to the following equation:

$$\frac{dy}{dx} = \sum_{i=1}^{2} \sum_{j=1}^{2} d_i w_{ji} a_j f'(w_{1i} f(a_1 x + b_1) + w_{2i} f(a_2 x + b_2) + c_i) f'(a_j x + b_j) \quad (3.13)$$

In our case we have $f(x) = ReLU(x) = max(0, x)$, if we approximate at 0 since ReLU are not differentiable at 0, this leads to:

$$f'(x) = \frac{max(0, x)}{x} = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases} \quad (3.14)$$

We can now see that the notion of blocked path appears. Indeed, we can see that Equation 3.13 is the sum of 4 different terms, each of which has the following form:

$$d_i w_{ji} a_j f'(w_{1i} f(a_1 x + b_1) + w_{2i} f(a_2 x + b_2) + c_i) f'(a_j x + b_j)$$

Let's discuss this equation term by term:

- First we have:

$$f'(w_{1i} f(a_1 x + b_1) + w_{2i} f(a_2 x + b_2) + c_i) = \begin{cases} 1, & \text{if } m_i \text{ is active} \\ 0, & \text{else} \end{cases}$$

- Similarly, we have:

$$f'(a_j x + b_j) = \begin{cases} 1, & \text{if } n_j \text{ is active} \\ 0, & \text{else} \end{cases}$$

- Finally, we have:

$$d_i w_{ji} a_j$$

  which corresponds to the product of the weights on the path

$$x \rightarrow n_j \rightarrow m_i \rightarrow y$$

One can now see how Equation 3.12 is obtained. In fact, when using this formula, the importance of an input is the sum of the weights products for all the paths leading to the output (i.e. non-blocked paths). A problem clearly appears even in the case of a single layer (shown on Figure 3.10). If there are two inputs and one output, the importance of each input will be given by its weight but then the input is not taken into account. For example :

$$input_1 = 10000, input_2 = 10, w_1 = 0.1, w_2 = 1$$

will lead to

$$Imp(n_1) = 0.1, Imp(n_2) = 1$$

even though neuron n_1 affected the output much more than neuron n_2 in this case. This can be extended to multiple layer networks and shows one of the limitation of the method with $ReLU$ activation functions.

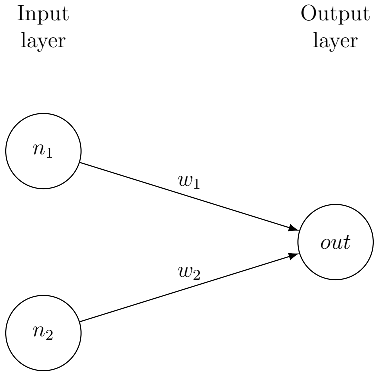

Figure 3.10: Graphical representation of a single layer network

3.2.3 Formulae comparison

The methods proposed in Subsection 3.2.1 and 3.2.2 will now be compared. As we will see, they are in fact not that different from one another. Indeed, we have:

- For the importance backpropagation method:

$$Total_x = \sum_{i=1}^{k} f(Out(n_i) * w_{ix}) \tag{3.4}$$

$$Imp(n_i) = \sum_{x=1}^{l} \frac{f(Out(n_i) * w_{ix}) * Imp(m_x)}{Total_x} \tag{3.5}$$

If we merge equations 3.4 and 3.5, we obtain:

$$Imp(n_i) = \sum_{x=1}^{l} \frac{f(Out(n_i) * w_{ix}) * Imp(m_x)}{\sum_{y=1}^{k} f(Out(n_y) * w_{yx})} \tag{3.15}$$

- For the derivative method:

$$\frac{dfinal_k}{dn_i} = \sum_{x=1}^{l} \frac{dfinal_k}{dm_x} * \frac{dm_x}{dn_i} = \sum_{x=1}^{l} \frac{dfinal_k}{dm_x} * w_{ix} * f'(\sum_{y=1}^{k} Out(n_y) * w_{yx} + Bias(m_x)) \tag{3.11}$$

Which can be rewritten as (by letting $\frac{dfinal_k}{dx_i} = Imp(x_i)$):

$$Imp(n_i) = \sum_{x=1}^{l} Imp(m_x) * w_{ix} * f'(\sum_{y=1}^{k} Out(n_y) * w_{yx} + Bias(m_x)) \quad (3.16)$$

If we use *ReLU* activation functions, with the definition of $f'(x) = \frac{f(x)}{x}$ we finally have:

$$Imp(n_i) = \sum_{x=1}^{l} \frac{Imp(m_x) * w_{ix} * f(\sum_{y=1}^{k} Out(n_y) * w_{yx} + Bias(m_x))}{\sum_{y=1}^{k} Out(n_y) * w_{yx} + Bias(m_x)} \quad (3.17)$$

As we can see, equations 3.15 and 3.17 have some common terms. In fact, in both cases the sum of inputs entering one neuron m_x appears at the denominator, except that it goes through the ReLU function in the case of the non-derivative formula. Similarly, the weights linking neuron n_i to those of the next layer appear at the numerator of both formulae. In one case, the output of neuron n_i is used whereas it doesn't appear in the other case, which is the main difference between the two methods. The similarity between both formulae can be used to motivate the usage of the importance back-propagation formula since it is very close from the derivative formula which is well motivated intuitively.

3.3 Mixed method (B.P. + I.L.)

In this Section we will propose a way to merge the invisible layer (Section 3.1) and backpropagation (Section 3.2) methods and describe the pros and cons of doing so.

Principle. Merging the two methods is actually straightforward, we just have to create a network with an invisible layer (as was shown on Figure 3.2) and train it. We then simply consider the input of the network as being the output of the invisible layer when running the backpropagation algorithm. By doing this, we simply multiply each feature importance of the back-propagation algorithm by the corresponding invisible layer's weight. Note that we must not take the absolute value of the invisible layer's weights in this case. In fact, if the invisible layer has some negative weights, the inputs which are taken into account in the back-propagation formula will be reversed, thus leading to incorrect importance back-propagation. Note that we could have tried to merge the algorithms by averaging the normalised importances, however that would require training two different networks. Indeed, a network should be trained without invisible layer in order to have unbiased B.P. importances. The invisible layer influences how the inputs are given through the network, it is thus not possible to obtain raw B.P. importances from such a network. Thus we decided that it makes more sense to merge the two techniques as explained above, this is also more similar to what happens inside the network.

pros. As it is often the case in machine learning, combining methods give improved results. This is no exception, the "mixed method" has given us the best results amongst all other methods. Furthermore, this method provides an invisible layer, leading to more control and easier regularisation.

cons. The network needs to have the invisible layer while being trained. It is thus not possible to compute feature importances this way on a pre-trained network as opposed to the back-propagation method alone. Also, the back-propagation formula has to be run in order to get feature importances. There is thus a computation time, as opposed to the invisible layer method alone for which the feature importances are immediately available after training.

Finally, note that we lose the ability to select redundant features due to the regularisation. Depending the problem, this can either be a pro or a con, although even in the *all-relevant* case, the regularisation can be adapted such that the redundant features are detected correctly.

3.4 Feature swapping techniques (F.S.)

In this section we introduce a general feature selection method. We will start by explaining the principles behind it and then talk about the limitations of this method for artificial neural networks. We will also propose an enhancement linked with Section 3.1.

3.4.1 Principles

The idea behind the features swapping technique is to assess the importance of each feature by looking at its effect on the prediction. This is done by perturbing a feature at test time and looking at how it impacts the error, Algorithm IV details this process.

Algorithm IV. *General procedure for feature selection using variable permutations methods.*

1. **Train model.** Train the model with a learning set in order to minimise an error term (cross-entropy, mean square error, ...).
2. **Assess performance.** Feed a test set to the network and predict the output then assess the performance of the model. This is done according to the error criteria (in our case we used MSE and accuracy for regression and classification problems respectively).
3. **Feature pass.**
   (a) Replace an input feature with some random noise generated following N(0,1) (for all test samples).This way of proceeding is correct only because inputs have been normalised beforehand.

(b) Feed the modified test set to the network and predict the output.

(c) Assess the performance degradation based on the error criteria.

$$\begin{cases} degradation \nearrow, & \text{if MSE} \nearrow \text{or accuracy} \searrow \\ degradation \searrow, & \text{if MSE} \searrow \text{or accuracy} \nearrow \end{cases}$$

4. **Evaluate importances.** Repeat step **3.** multiple times for each input feature and rank them according to the average performance degradation. The higher the degradation the more informative the feature.

3.4.2 Weakness

The main problem of this technique is that it is computationally heavy for high dimensional input spaces. Indeed, if K passes are done for each feature, the number of predictions required is:

$$Input\ dimension * nbr\ test\ samples * K$$

Thus it becomes quite lengthy to use for high input dimensions. In our case with 5000 input features and 3000 test samples we only used one pass per feature to carry out our tests. Indeed, we tried multiple passes per feature but the performance gain was negligible compared to the computation time increase.

3.4.3 Enhancement

As explained in Section 3.1, introducing an invisible layer has proved to increase network's predictions performances in the case of noisy inputs (as shown in Figure 3.4). Therefore, using it for feature swapping seems like a good idea. Thus if we want to improve the performances of the algorithm, we can add an invisible layer to the network and train it with a regularisation term.

3.5 Comparison and testing

This section aims at comparing the different methods that have been proposed. In particular Subsection 3.5.1 will analyse the behaviour of each algorithm on multiple artificial datasets whereas Subsection 3.5.2 will discuss the effect of the network architecture on the algorithms. We will then sum up the whole chapter in Subsection 3.5.3 by giving a table which covers the main characteristics of each method.

3.5.1 Results on artificial datasets

3.5.1.1 Methodology

Since there are lots of parameters to tune in neural networks, especially when regularisation is introduced, it is necessary to define a testing methodology. In our case, we decided to carry out every test with a neural network containing 4 hidden layers of 500 neurons each. This was chosen arbitrarily so that the network isn't too shallow. This way we hope to prove that the methods we proposed are viable even with deep complex networks. Now that we have explained how we chose our architecture, we need to tackle the "invisible layer regularisation" problem. Remember, we have :

$$cost = f(output_1, ..., output_n; real\_outputs) + \alpha * \frac{\sum_{i=1}^{n} |i_n|}{n} \tag{3.2}$$

We thus need to find the optimal α, which depends on the network's architecture as well as on the dataset's nature (noisy or not). Since it is impossible to find a priori, we used a "validation technique". Thus we trained networks with different regularisation parameters (defined on Table 3.1) on a training set and kept the one that gave the best accuracy/MSE on a validation set (which was generated the same way as the training set). We then use the α found to train the networks (with the same training set) on which we assess our algorithms. Note that we could have used cross-validation to obtain better results but found it useless since we were able to generate as many validation samples as wanted due to the artificial nature of the datasets.

	Tested values						
α	0.0	0.2	2.0	5.0	8.0	13.0	20.0

Table 3.1: Values of parameter α tested in order to find the optimum.

Datasets and algorithms used. We decided to analyse the following algorithms on multiple datasets :

- **I.L.** The regularised invisible layer algorithm.
- **B.P.** The importance back-propagation algorithm. For ease of read we do not give detailed results using the derivative formula since they were systematically worse than with the importance back-propagation formula. In fact, they were not even comparable as the derivative back-propagation algorithm never gave results exceeding an AUPR of 0.006 (whatever the dataset). As we will see, this is abysmal compared to the other proposed techniques.
- **B.P. + I.L.** The mixed method with regularisation
- **F.S. + I.L.** The feature swapping method combined with the invisible layer. We didn't look at F.S. without the invisible layer since the regularisation enhances accuracy so much that F.S. alone is always surpassed by other algorithms (even B.P. alone).
- **Random forests.** The benchmark algorithm we use to assess the performances of our different methods. All of our random forests are trained with 5000 trees, a maximum depth of 30 and $K = log2$.

These algorithms were used on three different artificial datasets which we name here:

- **Classification.** In which we will look at a noisy hypercube classification problem. (generated by Scikit-learn [skl, b])
- **Regression.** Where we will analyse a noisy linear regression problem. (generated by Scikit-learn [skl, c])
- **Tree classification.** Where we generated a classification dataset as proposed in [Li et al., 2015].

3.5.1.2 Classification

The first dataset we will discuss consists in a binary classification problem and is defined in Appendix A. We analysed the results obtained with different level of noise, with 2000 training samples and 1000 validation samples. Remember that the higher the AUPR, the better the feature selection. Multiple things owe to be mentioned here (see Table 3.2):

- First, notice that the α parameter plays a huge role, the noisier the dataset, the higher the gains from regularisation. Indeed we can see that for low noise level (25 useless features), the best α found was only equal to 2.0 and didn't improve accuracy that much compared to methods without regularisation (B.P. and RF). However we see in the three other cases (475, 2475 and 4975 useless features respectively) that the best α found is bigger (equal to 8.0), makes a big difference in accuracy and gives a boost to the AUPR.
- Second, random forests perform better than the back-propagation (B.P.) algorithm be it in terms of accuracy or AUPR. However, as stated above, once we add regularisation all the neural network algorithms have better performances than random forests, at least on noisier datasets.
- Third, even though the feature swapping has been mixed with the invisible layer (F.L. + I.L.) to have better accuracies, in terms of feature selection this algorithm performs poorly compared to others in these settings.
- Finally, we can see how the mixed (B.P. + I.L.) algorithm always gives (or ties for) the best feature selection. We have found this algorithm to be the most robust amongst all of them.

25 useful	inputs	I.L.	B.P.	B.P. + I.L.	F.S. + I.L.	RF
50 Total inputs	AUPR	**1.00**	**1.00**	**1.00**	**0.99**	**1.00**
	accuracy	0.97	0.95	0.97	0.96	0.90
	α	2	0	2	2	0
500 Total inputs	AUPR	**1.00**	0.88	**1.00**	0.93	0.98
	accuracy	0.96	0.78	0.94	0.94	0.83
	α	8	0	8	8	0
2500 Total inputs	AUPR	0.91	0.63	**0.98**	0.81	0.71
	accuracy	0.92	0.69	0.92	0.91	0.76
	α	8	0	8	8	0
5000 Total inputs	AUPR	**0.77**	0.61	**0.77**	0.59	0.68
	accuracy	0.89	0.64	0.87	0.88	0.68
	α	8	0	8	8	0

Table 3.2: Comparison of the algorithms on different hypercube binary classification datasets.

We thought it would also be of some interest to look at the dynamics of these algorithms with respect to training iterations. Note that since there is no such thing as training iterations for random forests, we decided to train them with a number of trees equal to $\frac{training\ iterations}{10}$. The “training dynamics” (for the 2475 non-informative features setting) are presented on Figure 3.11, as a reminder:

- First graph represents the evolution of accuracy with respect to training iterations.
- Second graph represents the evolution of AUPR with respect to training iterations.
- Third graph represents the evolution of feature importances with respect to training iterations. Dashed line represents the average feature importance of useless features whereas filled line represents the average importance of useful ones. Thus the best algorithm should separate dashed and filled lines as mush as possible.

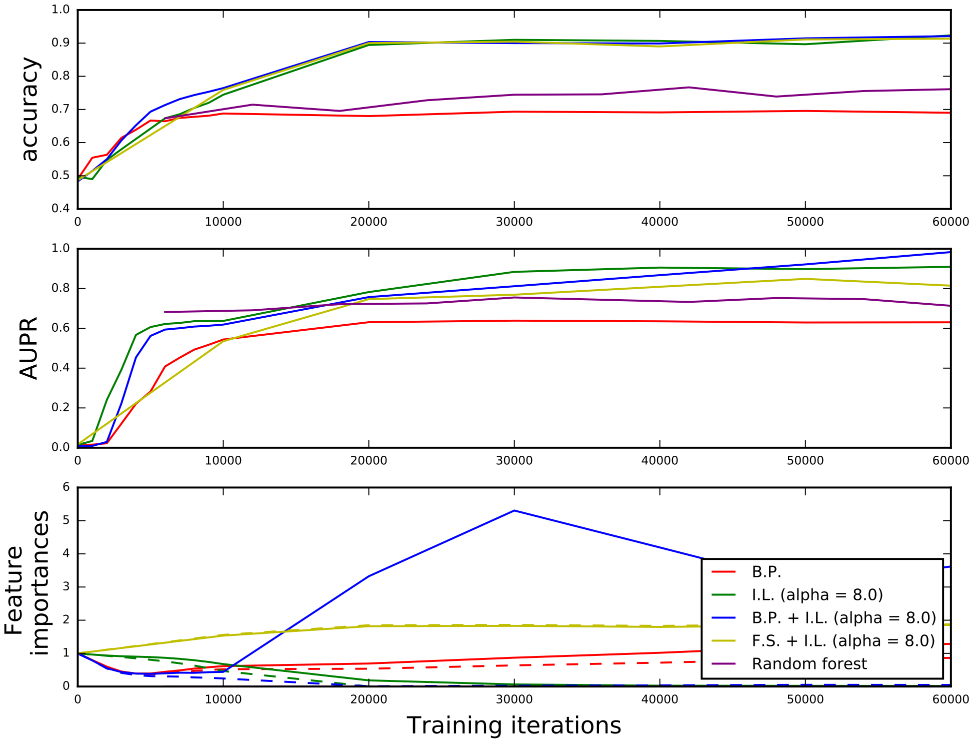

Figure 3.11: Accuracy, AUPR and weights evolution of a network for different algorithms. These are the dynamics of the results obtained in the third row of Table 3.2.

3.5.1.3 Regression

The second dataset we will look at is a simple linear regression problem. Indeed we generated 2000 training samples and 1000 validation samples using the following equation:

$$y = \sum_{i=1}^{n} w_i * x_i \tag{3.18}$$

where n is the number of useful features (25 in our case) and where each w_i is drawn uniformly in $[0, ..., 100]$. Non-relevant features have been added in order to introduce noise into the network. These features are generated the same way as for the classification problem (i.e. with different noise types). The testing methodology remains the same as for classification and results are presented on Table 3.3.

25 useful	inputs	I.L.	B.P.	B.P. + I.L.	F.S. + I.L.	RF
50 Total inputs	AUPR	0.98	0.98	**0.99**	**1.00**	0.99
	MSE	0.01	0.32	0.01	0.02	1.11
	α	8	0	8	8	0
500 Total inputs	AUPR	**1.00**	0.86	**0.99**	0.96	0.86
	MSE	0.01	0.49	0.01	0.01	1.03
	α	13	0	13	13	0
2500 Total inputs	AUPR	**0.95**	0.88	**0.94**	**0.96**	0.83
	MSE	0.03	0.78	0.02	0.03	0.98
	α	13	0	13	13	0
5000 Total inputs	AUPR	0.98	0.87	**1.00**	**1.00**	0.80
	MSE	0.04	0.89	0.04	0.04	1.01
	α	13	0	13	13	0

Table 3.3: Comparison of the algorithms on different linear regression datasets.

As we can see, neural networks tend to be very accurate in this setting to the point of surpassing random forests even free of regularisation. However, note that this problem setting should be unfavourable for random forests. Nevertheless, this is a very interesting result especially for noisier datasets where B.P. algorithm has an AUPR of 0.874 against 0.801 for random forests. We also see that the regularisation helps keeping a really low MSE and thus the F.S. + I.L. works really well here. However the B.P. + I.L. technique also works really well and remains a sure bet. Finally notice that the regularisation pattern is the same than for classification, for lower noise level, lower regularisation seems to work better (which seems logical).

3.5.1.4 Tree classification

The last dataset we will be looking at in this Section has been proposed by [Li et al., 2015]. It has 9 input variables which are defined below and is generated following a decision tree which is shown hereunder on Figure 3.12.

- $x_0 \sim U(-1,1)$ Informative,
- $x_1 = -x_0$ Informative,
- $x_2 \sim U(-1,1)$ Informative,
- $x_3 \sim U(-1,1)$ Informative,
- $x_4 \sim N(0,1)$ Non informative,
- $x_5 \sim L(0,1)$ Non informative,
- $x_6 \sim U(0,1)$ Non informative,
- $x_7 \sim U(-1,0)$ Non informative,
- $x_8 \sim U(-1,1)$ Non informative,

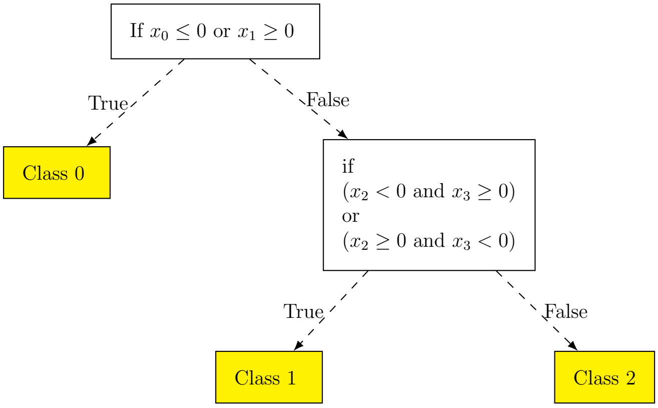
If $x_0 \leq 0$ or $x_1 \geq 0$
True
False
Class 0
if
($x_2 < 0$ and $x_3 \geq 0$)
or
($x_2 \geq 0$ and $x_3 < 0$)
True
False
Class 1
Class 2

Figure 3.12: Rules defining the dataset generation (taken from [Li et al., 2015]).

We used 350 training samples and 150 validation samples, we did not use the exact same dataset as [Li et al., 2015] since they generated 9000 samples which makes the problem too easy in our opinion. The best α was found to be 2.0, we show the features importances for each algorithm on Table 3.4. We can observe that due to the simple nature of the problem, feature selection is done correctly. As mentioned before in Section 3.1, the algorithms with I.L. are only able to select one of x_1 or x_0 since they are redundant.

Variable importances	I.L.	B.P.	B.P. + I.L.	F.S. + I.L.	Random forests
x_0	**1.32e−2**	**6.46**	**0.94**	**0.84**	**0.33**
x_1	1.6e−6	**6.42**	0.02	0.43	**0.31**
x_2	**1.01e−2**	**2.22**	**0.31**	**0.59**	**0.06**
x_3	**1.06e−2**	**2.64**	**0.32**	**0.56**	**0.07**
x_4	2.05e−5	0.92	0.02	0.43	0.04
x_5	1.07e−5	0.92	0.01	0.42	0.04
x_6	8.6e−6	1.5	0.01	0.44	0.04
x_7	1e−5	1.2	0.02	0.43	0.05
x_8	1.35e−5	1.29	0.01	0.43	0.04

Table 3.4: Feature importances found by different algorithms on dataset proposed by [Li et al., 2015].

3.5.2 Architecture effect analysis

Now that the algorithms have been compared on different datasets we will analyse the neural architecture effect. To do this we will use the hypercube classification dataset with 25 informative variables and 2475 non-informative variables (this is the same dataset that was used for Figure 3.11). We still use a validation technique to chose the best regularisation parameter and set the number of neurons per layer to 500. Table 3.5 presents the results obtained with 1, 2, 3, 4 and 5 hidden layers respectively, where each network has been trained on 375 epochs per hidden layer (i.e. for n hidden layers, the network runs $n * 15000$ training iterations on batches of size 50 and there are 2000 training samples). As we can see, feature selection performance is tightly linked to the accuracy of the network. This property is interesting since it allows us to optimise the architecture for feature selection by cross-validation on the accuracy. In fact, this tells us that if one wants to create a network to analyse the features (regardless predictive performances), one should still create the best network possible (predictive performance wise) and analyse its features. Also note that the architecture plays an important role; if the network is too small, it won't be able to learn correctly and performances will be degraded. However, the network must not bee too complex either at the risk of losing accuracy due to overfitting. The overfitting effect can be seen with 5 hidden layers, as performances start to decrease be it in terms of accuracy or AUPR. All in all we show here that there is a balance to be found in the architecture for feature selection, we also show that the accuracy is a good tool to tune the architecture. Finally, we can see that the B.P. + I.L. is once again the most stable algorithm and gives really good performances if the network is complex enough.

25 useful	inputs	I.L.	B.P.	B.P. + I.L.	F.S. + I.L.	RF
1 hidden layer(s)	AUPR	0.62	0.59	**0.62**	0.53	**0.71**
	acc.	0.76	0.62	0.76	0.76	0.76
	α	20	0	20	20	0
2 hidden layer(s)	AUPR	**0.92**	0.61	**0.89**	0.65	0.71
	acc.	0.82	0.61	0.83	0.78	0.76
	α	5	0	5	5	0
3 hidden layer(s)	AUPR	0.88	0.61	**0.94**	0.78	0.71
	acc.	0.85	0.65	0.85	0.84	0.76
	α	20	0	20	20	0
4 hidden layer(s)	AUPR	0.91	0.63	**0.98**	0.81	0.71
	acc.	0.92	0.69	0.92	0.91	0.76
	α	8	0	8	8	0
5 hidden layer(s)	AUPR	0.93	0.61	**0.95**	0.84	0.71
	acc.	0.90	0.62	0.88	0.88	0.76
	α	20	0	20	20	0

Table 3.5: Comparison of the algorithms with different architectures for 25 informative inputs and 2475 non informative inputs (hypercube classification dataset).

3.5.3 Pros and cons of each method

In this Subsection we will summarise all of the algorithms proposed in a single table. The color indicators shown in Figure 3.13 will be used.

Color indicators	
Best	🟩
...	🟩
...	🟨
Worst	🟥

Figure 3.13: Color indicators

Five main characteristics will be studied for each method:

1. **Performance.** Represents how well the algorithm selects useful features.

2. **Plug & Play.** Can the algorithm be used on a pre-trained network, or does it require to create a new network and restart all the learning process in order to derive feature importances ?

3. **Finds redundancy.** Is the algorithm **in its current state** able to find redundant variables or not ?

4. **Computation time.** Represents the time taken by the algorithm to predict feature importances after training.

5. **Improves prediction performance.** Can the regularisation increase accuracy or reduce MSE thanks to the invisible layer ?

The results are shown hereunder in Table 3.6.

Method	Performance	Plug & Play	Finds redundancy	Computation time	Improves prediction performances (can be regularised)
Invisible layer (I.L.)	🟩	🟥	🟥	🟩	🟩
Backpropagation (B.P.)	🟧	🟩	🟩	🟧	🟥
Feature swapping (F.S.)	🟥	🟩	🟩	🟥	🟥
F.S. + I.L.	🟧	🟥	🟥	🟥	🟩
B.P. + I.L.	🟩	🟥	🟥	🟧	🟩

Table 3.6: Summary table of algorithm's main characteristics

3.6 Conclusion

Multiple algorithms have been presented with their advantages and drawbacks, the choice of which to use is thus situation dependant. If the goal is to select features on a un-noised dataset or to use a pretrained network, then the B.P. algorithm should be used. Otherwise using the B.P + I.L. technique is the way to go most of the time. It must not be forgotten that as shown in Subsection 4.2.2 neural architecture plays a big role. Indeed, results can vary greatly with the number of hidden layers/neurons per layer. However we showed that since the higher the accuracy the better the feature selection, this problem can be addressed by using cross-validation to find a near optimal architecture. Finally, we showed that the computation time of the B.P and B.P + I.L. algorithms is of the order of an epoch, which is really not a huge deal compared to the training time. Therefore, only the swapping techniques suffer from their computation time.

It is also very important to remember the extensions given for each algorithm.

- First, remember that the regularisation can be modified as explained in Section 3.1 according to the problem. For example, it is sometimes considered useful to detect redundant features but can also be detrimental. The I.L. should be seen as a powerful tool to add some regularisation on the network such that it can enhance other methods.

- Second, we have shown in Subsection 3.2.1 that the B.P. technique could be used for other things than feature selection. For example, since the method is quite flexible, we showed that by changing a function in the formula we would be able to prune the neural network without hurting accuracy. Also as stated, multiple algorithm initialisation methods could be imagined and we have given clues on what could be changed to the current algorithm in order to further enhance the results. As an example we have given another way to initialise the B.P. algorithm in the case of binary classification with Equation 3.7, which would likely result in enhanced performances. However the general initialisation method that we provided was shown to work quite well for all the problems we tackled.

Chapter 4

Application : Inference of Gene Regulation

In this Chapter we look at a bioinformatic problem for which feature selection is required. In Section 4.1, we state the problem and talk about the current state of the art results. During this explanation we will see that the main problem can be tackled in two different ways. The first option is detailed in Section 4.2 and consists in using single-output regression models, whereas the second option uses multi-output regression models and is discussed in Section 4.3. Finally Section 4.4 summarises the Chapter and shows what can be achieved with our proposed methods compared to the state of the art for this specific problem.

4.1 Posing the problem

In this section, we start by detailing the problem and give motivations on why it is important. We then talk about the different datasets that exist as well as explain our choice for which to investigate. The next step is to give an insight on the two main ways that the problem can be solved with machine learning techniques (i.e. single-output vs. multi-output regression). Finally we discuss the state of the art algorithms and show their performances on our chosen dataset.

4.1.1 General problem

Biology has always been a very interesting field, and genetics in particular. Nowadays with the advances of genome sequencing the hard part is not to know what we are made-off anymore, but rather to understand it. With more and more data becoming available, computer analysis is becoming more and more necessary to understand the genome. The problem we will be looking at in this chapter is called gene network inference. It is equivalent to building a directed graph with a given set of genes, where a gene being linked to another means it influences it. To this end, we need to measure some characteristics of the genes, in this case we use “gene expression”. It corresponds to the proportion of a given gene present in the RNA form (transcripted DNA) which can be seen as representing the proportion of that gene in a tissue at a given time compared to others.

4.1.2 Posing the methodology

Now that we have explained what "gene expression" represents, we can use it in order to tackle our initial problem : **Gene network inference**. Indeed, the datasets consist of n samples of m measured gene expressions each. We thus need to define a method in order to assess our feature selection against the real directed graph. To do this we create an m by m matrix (which will be referred to as the **"importances matrix"** from now on) where element m_{ij} corresponds to the importance of gene i to predict gene j.

Once the importances have been computed for each pair of genes, we use that measure in order to rank the links between variables. Indeed, given a threshold t, we have:

- if $m_{ij} \geq t$ then we consider that $gene_i \rightarrow gene_j$ exists (i.e. m_{ij} is a positive)
- if $m_{ij} < t$ then m_{ij} then we consider that $gene_i \rightarrow gene_j$ doesn't exist (i.e. m_{ij} is a negative)

We then check the real graph (i.e. ground truth) to see if the link $gene_i \rightarrow gene_j$ exists or not. This gives us the confusion matrix for a given threshold. Repeating this for all possible thresholds we can compute the AUPR and assess our algorithm (as explained in Chapter 2).

4.1.3 Existing datasets

The gene network inference problem has been proposed for multiple contests throughout the years. The datasets were originally in silico (i.e. artificial datasets) but evolved to become in vivo (i.e. real datasets). The different contests were the following:

1. **DREAM2** (2007) in silico.
2. **DREAM3** (2008) in silico.
3. **DREAM4** (2009) in silico.
4. **DREAM5** (2010) in vivo and in silico.

We decided to tackle the DREAM4 datasets, since it is one of the most recent problems and is more friendly than the DREAM5 contest. Indeed, the DREAM5 datasets are about micro-organisms and use more than 1500 genes, whereas the DREAM4 datasets' networks are of size 10 and 100 respectively. The DREAM challenges and especially DREAM4 are discussed and presented in the following papers: [Marbach et al., 2009], [Stolovitzky et al., 2009] and [Stolovitzk et al., 2007].

Within the DREAM4 contest we chose to tackle the 100 multifactorial genes networks to prove that neural network can work in non-trivial settings. In fact, five different multifactorial networks are given in the challenge and we tested our algorithms on each of them. Note that a potential extension to work with knockdown and knock-up data will be briefly discussed in Section 4.4, although since this thesis is not about the DREAM4 challenge itself it has not been analysed here. Indeed, our goal is to prove that the feature selection methods described in Chapter 3 are generic and viable enough to be used in different problem settings. This is a good problem to "stress-test" our methods since the datasets consist of 100 genes that have been perturbed (multi-factorial), and we only possess 100 samples. This is thus a problem that should a priori highly favour random forests over neural networks since ANN are known to require lots of data. If we manage to achieve similar performances than random forests on such unfavourable datasets, it seems promising for the viability of our methods.

4.1.4 General approaches

As mentioned before, there are two ways to use feature selection in order to compute the importances matrix (of size m by m).

- The first one is to use single-output regression models and the procedure is the following:

    1. Normalise each gene.
    2. $\forall i \in [1, ..., m]$, make $gene_i$ the prediction target and use the remaining $m - 1$ genes as input variables to learn a model. Then, the feature importances of the given neural network correspond to column i of the importances matrix (where the element of the diagonal is set to be 0 since a gene can't be used to predict itself).

    We will look at the results obtained while using our algorithms with this method in Section 4.2.

- The other one is to use multi-output regression models. In this case, we also need to **normalise** each gene first but then we only need to train a single regression model. We look at the importances of each input feature to predict a single of the outputs. Importance of $input_i$ to predict $output_j$ gives the element m_{ij} of the importances matrix. Once every m_{ij} has been computed, we set all m_{ii} to 0 to be sure that a gene can't be considered informative for itself.

4.1.5 State of the art

The DREAM4 challenge was won using single output random forests (GENIE3 method) as proposed in [Huynh-Thu et al., 2010]. The results obtained with this method on the five different multi-factorial datasets are shown in Table 4.1. In order to make a fair assessment of our methods, we decided to rerun the GENIE3 algorithm using our own scripts and use those results as a comparison for our algorithms. By doing this, we ensure that the AUPR of each algorithm is computed the exact same way. As can be seen on Table 4.1, the results obtained with our scripts are almost identical to those presented in the paper.

	AUPR				
	Set 1	Set 2	Set 3	Set 4	Set 5
GENIE3 results (as presented in the paper)	0.154	0.155	0.231	0.208	0.197
GENIE3 results (as obtained with our script)	0.152	0.154	0.228	0.209	0.195

Table 4.1: GENIE3 results on DREAM4 challenge (taken from [Huynh-Thu et al., 2010]) and GENIE3 results obtained with our script.

4.2 Single-output approach

This Section aims at using the GENIE3 methodology with neural networks. To better understand where the final results come from, we structured the Subsections as our experiments were carried out. This way, we show the problems we encountered and give our proposed solutions, leading to the final iteration of our algorithm which embeds a few tricks to tackle the problems encountered.

1. **First try and what not to do.** We start by explaining how we first tried to tackle the problem in order to see whether or not our algorithms could be viable. As we will see, results were decent but could be easily improved.

2. **Architecture analysis, brute force approach.** We look at a multitude of different architectures and regularisation parameters and discuss their influence on the AUPR of the dataset 1.

3. **Cross validation technique.** A general procedure is given and tested on all of the 5 datasets.

4. **Conclusion.**

4.2.1 First try and what not to do.

There is unfortunately no standard best architecture for neural network. It is of course problem-dependant, therefore as a first test, we decided to use network architecture that seemed appropriate for the DREAM4 datasets. This led us to use networks of size [99:300:300:300:300:1] as represented on Figure 4.1. As we can see there is an invisible layer in the network's architecture, this is because we decided to only look at the I.L. and B.P. + I.L. methods for this part. Since the B.P. + I.L. proved to be the best overall it seemed like the most well-fitted algorithm to tackle such kind of problem. In fact, due to the low number of samples, we were never able to achieve a good MSE on the problem (this will be discussed in Subsection 4.2.3). This would cause the F.S. algorithm to give poor results. Since using the B.P. + I.L. method allows us to get the feature importances from the I.L. without any additional computations, we also decided to peak at the results for the I.L. alone. As we will see, this allows us to further prove the usefulness of the back-propagation method.

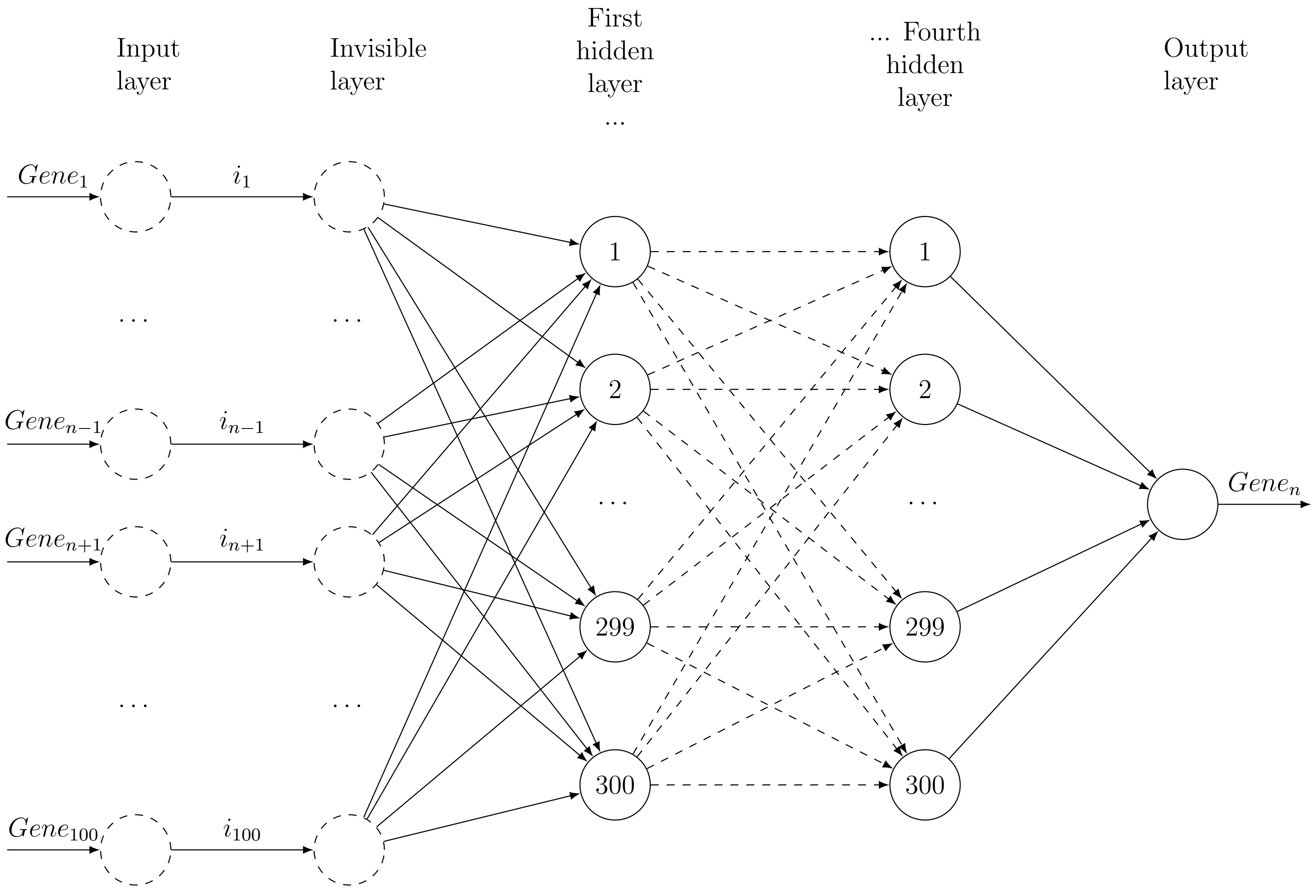

Figure 4.1: Network built for $gene_n$ prediction. Features importances will give column n of the importances matrix.

Once the network's architecture is defined, the problem of the regularisation parameter still remains. Indeed, we have shown in Section 3.5 that the optimal α is problem-dependant. Thus the α parameter should be adapted for the 5 different DREAM4 datasets. To this end, for each dataset, we first trained a network for a given gene, optimised the α parameter by cross-validation for that particular case and used this value for all other genes. Of course, this method is not optimal since there are 100 different networks that must be trained per dataset (one for each gene), thus maybe the optimal α for a given *gene* is not the best over all the networks. Still, we decided to test the technique and obtained the following results (see Table 4.2).

		I.L.	B.P. + I.L.	GENIE3	Maximum difference
Set 1	AUPR	0.1396	0.1064	**0.152**	-0.0124
	α		0	x	
Set 2	AUPR	0.099	0.0977	**0.154**	-0.055
	α		0	x	
Set 3	AUPR	0.1813	0.1939	**0.228**	-0.0341
	α		8	x	
Set 4	AUPR	0.1927	0.205	**0.209**	-0.004
	α		8	x	
Set 5	AUPR	0.1692	0.1734	**0.195**	-0.0216
	α		2	x	

Table 4.2: Results obtained for the 5 DREAM4 datasets. These were obtained using 100 different networks of size [99:300:300:300:300:1] per dataset. The α parameter obtained has been optimised by cross-validation on a single gene chosen randomly.

As can be seen in Table 4.2, the first results we obtained with neural networks were promising, even though worse than the GENIE3 method. From there, we decided to carry out a deep analysis on the impact of the architecture of the network as well as the regularisation parameter.

4.2.2 Architecture analysis, brute force approach.

As suggested in Section 3.5, neural architecture plays a big role in the performances of a network. In order to see what can be achieved with a near-optimal architecture, we decided to analyse the first of the 5 datasets with a multitude of neural architectures. We chose to train networks with a combination of the following parameters :

- Number of hidden layers : [1,2,3,4]
- Number of neurons per layer : [1,10,100,300]
- α parameter : [0,1,4,8,13]

By training 100 networks for each of the 80 (i.e. $4 * 4 * 5$) possible architectures, we were able to compute the $AUPR$ of each architecture for the I.L. and B.P. + I.L. algorithms. The results are shown on Figure 4.2 and Figure 4.3 respectively.

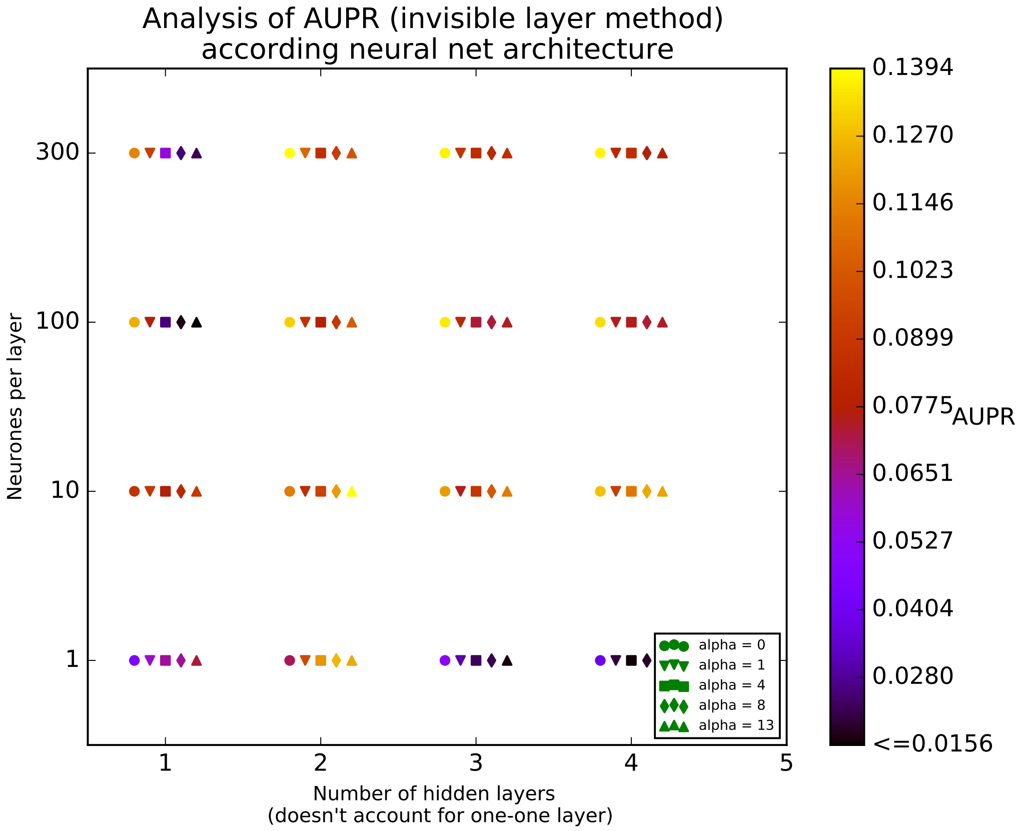

Figure 4.2: AUPR score computed with the I.L. technique for each of the possible network architecture.

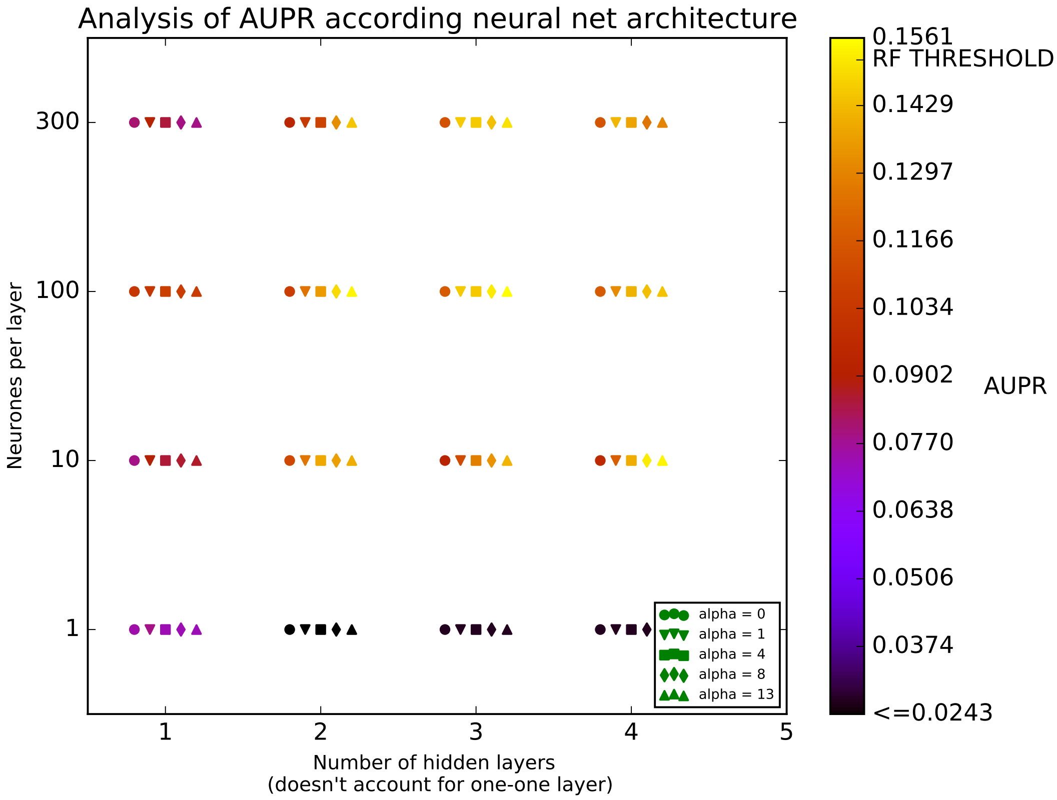

Figure 4.3: AUPR score computed with the I.L. + B.P. technique for each of the possible network architecture.

Multiple things owe to be highlighted here :

- Be it for the I.L. or B.P. + I.L. method, we can clearly see that adding layers and neurons improves performance. In fact, the same remark as in Subsection 4.2.2 can be made: the optimal network is neither too shallow nor too complex. The right middle has to be found. This proves the interest of using deep neural network over simple linear regression methods.

- **We can clearly see the usefulness of the backpropagation technique**. Indeed we have a maximum AUPR of 0.1561 for the B.P. + I.L. method whereas it is only of 0.1394 for the I.L. method. This difference was not as clearly apparent in Chapter 3's tests, however here it clearly motivates the usage of the B.P. technique.

- We can see an interesting pattern for the B.P. + I.L. method. Running the backpropagation algorithm seems to make the network more robust to regularisation. Indeed, we see that in most cases, the higher the regularisation the better the results. Whereas this is not at all the case for the I.L. method. Thus, the backpropagation can be seen as a way to reclaim the information lost at the first layer by looking at the deeper layers.

- We beat GENIE3 as can be seen with the **RF threshold** on Figure 4.3 method on this dataset if given the appropriate architecture.

Since we managed to beat the GENIE3 algorithm with one of the architecture, we decided to use the best architecture found (i.e. 3 hidden layers of 100 neurons each with $\alpha = 13$) on the 4 other datasets. The results obtained are summarised in Table 4.3

		I.L.	B.P. + I.L.	GENIE3	Maximum difference
Set 1	AUPR	0.075	**0.1575**	0.152	+0.0055
Set 2	AUPR	0.0923	0.1172	**0.154**	−0.0368
Set 3	AUPR	0.1656	0.1952	**0.228**	−0.0328
Set 4	AUPR	0.137	0.1741	**0.209**	−0.0319
Set 5	AUPR	0.0832	0.1553	**0.195**	−0.0397

Table 4.3: Results obtained for the 5 DREAM4 datasets. These were obtained using 100 different networks of size [99:100:100:100:1] per dataset, trained with $\alpha = 13$. This correspond to the architecture that was found to be near-optimal for dataset 1.

As we can see, with this architecture we obtain slightly better results on sets 2 and 3 but worse on sets 4 and 5. As expected, the biggest improvement is on set 1 which is the set on which the architecture was optimised. This leads us to the conclusion that even though the problems are quite similar, a single optimal architecture can't be found for all of them. The deep analysis carried out here is in fact not practical. Indeed, for some real problems there is no way to compute the AUPR since we do not have ground truth. Thus in Subsection 4.2.3, we propose a technique to find a decent architecture with a general method and show results on the 5 sets.

4.2.3 Cross validation technique

Since it is not possible to know a priori the best architecture for a given problem, we will use the same technique than we used to find the near-optimal α. To do this, we define multiple architectures which will be tested and we keep the one that gives us the best MSE which seems a reasonable proxy for the AUPR on the reconstruction of the network. The parameters tested are shown hereunder on Table 4.4 :

Number of hidden layers	Number of neurons per layer	α
2 3	50 150	0.5 3 9

Table 4.4: Possibilities for each of the three parameters to optimise.

Note that with so few parameters there are already $2*2*3 = 12$ possible combinations to cross validate. As we will see, this is already very computation heavy due to the cross validation technique used. Let us now detail the procedure to find the best combination, first we describe the optimal algorithm.

Algorithm VI. *Optimal method to cross-validate neural architecture using single output networks. k is the number of folds for the cross validation and n is the number of samples available.*

```
1. For every possible architecture:
   (a) For $i$ in $[1, ..., k]$ repeat the following steps:
       i. Choose the $i^{th}$ batch of $n/k$ samples as validation samples and the rest
          as training samples.
       ii. For each gene, repeat the following:
           A. Train network with the given training samples.
           B. Compute MSE on the validation samples.
       iii. Compute the average MSE over all the genes.
   (b) Compute the average MSE over all the cross-validation folds.
2. Store the global MSE for the given architecture.
```

We then keep the architecture with the lowest MSE overall as the best one. Note that we could adapt Algorithm VI to choose one architecture per gene. This is proposed in Algorithm VIbis. Using this technique the network has to be adapted depending on the target gene.

Algorithm VIbis. *Optimal method to cross-validate neural architecture per gene using single output networks. k is the number of folds for the cross validation and n is the number of samples available.*

```
1. For each gene repeat the following steps:
   (a) For each possible architecture:
       i. For $i$ in $[1, ..., k]$ repeat the following steps:
          A. Choose the $i^{th}$ batch of $n/k$ samples as validation samples and the
             rest as training samples.
          B. Train network with the given training samples.
          C. Compute MSE on the validation samples.
       ii. Compute the average MSE over all the cross validation folds for the given
           gene.
   (b) Keep the architecture that gives the best MSE for that gene.
```

The main drawback of the method proposed in Algorithm VI and Algorithm VIbis is the execution time. Indeed we have :

$$Number\ of\ networks = combinations * k * number\ of\ genes$$

which in our case with a 5fold cross validation leads to:

$$Number\ of\ networks = 12 * 5 * 100 = 6000$$

It is no surprise that training 6000 networks takes a long time. This scales in a linear fashion with respect to the number of folds. Since we needed to analyse 5 different datasets, the number of networks to train was simply off-putting. We thus decided to use a non optimal cross validation technique to reduce computation time. We still wanted to have the following properties:

- Each gene should be at least validated on once.
- Each sample should be at least validated on once.

Algorithm VII respects both of these properties and only needs to train m networks where m is the number of genes. Therefore, cross validation using this technique only requires training the same number of networks than to build the importance matrix for each possible combination. This means that the computation time compared to computing the importance matrix with a given architecture scales linearly with the number of combinations. Furthermore the computation time is independent of the number of folds, which leads to reasonable computation time.

Algorithm VII. *A Sub-optimal cross validation approach to find best neural architecture for a given dataset. k is the number of folds, n the number of samples and m the number of genes.*

```
1. For every possible architecture:
    (a) For $i$ in $[1, ..., k]$ repeat the following steps:
        i. Choose the $i^{th}$ batch of $n/k$ samples as validation samples and the rest as training samples.
        ii. For each gene in $i^{th}$ batch of $m/k$ genes, repeat the following:
            A. Train network with the given training samples.
            B. Compute MSE on the validation samples.
        iii. Compute the average MSE over all of these genes.
    (b) Compute the average MSE over all the genes.
2. Store the global MSE for the given architecture.
```

Using this technique, each gene is validated once and each sample is used once as a validation sample. The extreme case would be to do m folds and would lead to using n/m samples to validate each gene. As we can see, here we have a much more reasonable number of networks to train :

$$Number\ of\ networks = combinations * number\ of\ genes = 12 * 100$$

We used Algorithm VII with the parameters defined in Table 4.4 and obtained the average MSE for each dataset. Regarded in Table 4.5, MSE can be quite close for very different architectures (i.e. 0.73228 for 2 hidden layers and $\alpha = 3$ on set 3 versus 0.7349 for 3 hidden layers and $\alpha = 9$). This shows that this method has its flows and that heavier computations should be carried out if it is necessary to have a near-optimal solution.

			Average MSE over all genes				
Hidden layers	Neurons per layers	α	Set 1	Set 2	Set 3	Set 4	Set 5
2	50	0.5	0.88731	1.03605	0.92404	1.00245	1.13578
		3	0.89636	1.04840	0.96068	1.02438	1.14860
		9	0.90104	1.06240	0.99441	1.04230	1.16429
	150	0.5	0.92619	1.12586	0.78926	0.87968	1.10851
		3	0.83678	1.02342	**0.73228**	0.81336	1.03347
		9	**0.80512**	**0.98696**	0.73460	**0.80655**	**1.02330**
3	50	0.5	0.89334	1.05259	0.96079	1.03020	1.14104
		3	0.89747	1.05545	0.97848	1.03934	1.14677
		9	0.90115	1.06386	0.99062	1.04510	1.15991
	150	0.5	1.01454	1.23339	0.86516	0.96127	1.19750
		3	0.96126	1.18541	0.80744	0.89230	1.13567
		9	0.86565	1.07389	0.73490	0.82653	1.04447

Table 4.5: MSE computed with sub-optimal cross validation (Algorithm VII) for each architecture/dataset combination.

Finally, Table 4.6 shows the performance of feature selection using the best architecture found (as shown in Table 4.5). As we can see, performances are getting much closer to the ones of GENIE3 method even though very few architectures have been tested. It is also of great interest to note that we are able to select relevant features even with poor MSE. Indeed, Table 4.6 shows that our best results are obtained on dataset 5 even though the MSE for this dataset is worse than random guessing. Remember that each gene has been normalised to have a standard deviation of 1 and a mean of 0. Thus predicting 0 for each gene leads to an average MSE of 1.

		I.L.	B.P. + I.L.	GENIE3	Maximum difference
Set 1	AUPR	0.1353	0.1392	**0.152**	−0.0128
Set 2	AUPR	0.1177	0.1041	**0.154**	−0.0363
Set 3	AUPR	0.2014	0.2168	**0.228**	−0.0112
Set 4	AUPR	0.198	0.2046	**0.209**	−0.0044
Set 5	AUPR	0.1715	0.1933	**0.195**	−0.0017

Table 4.6: Results obtained for the 5 DREAM4 datasets. These were obtained using the best architecture found by sub-optimal cross validation (highlighted in bold on Table 4.5).

Now that we have a general algorithm to obtain a decent architecture without having to tune hyper-parameters by hand, we can further enhance the feature selection performance. Indeed, we noticed there was quite a bit of variation in the AUPR even when using the same method for models with the same architecture. To counter that, we decided to use an ensemble of models and average their feature importances. This method is given by Algorithm VIII.

Algorithm VIII. *An ensemble approach to enhance the feature selection performance with near-optimal neural architecture. Let k be the number of models to train, n the number of samples and m the number of genes.*

```
1. For $i$ in [1,...,k]:
2. Create importances matrix $i$
    (a) For $j$ in [1,...,m]:
        i. Train network with $gene_j$ as target
        ii. Compute features importance
        iii. Add those importances to the $j^{th}$ column of the importances matrix $i$
    (b) Save importances matrix $i$
3. Add all matrices together, this gives the final feature importances.
```

Using this method with $k = 5$ we managed to enhance the results of the best architecture even further. Our final results are shown in Table 4.7.

		I.L.	B.P. + I.L.	GENIE3	Maximum difference
Set 1	AUPR	0.1407	0.1419	**0.152**	−0.0101
Set 2	AUPR	0.1182	0.1072	**0.154**	−0.0358
Set 3	AUPR	0.2011	0.2174	**0.228**	−0.0106
Set 4	AUPR	0.1997	0.206	**0.209**	−0.003
Set 5	AUPR	0.1793	**0.1963**	0.195	+0.0013

Table 4.7: Results obtained for the 5 DREAM4 datasets. These were obtained using the Algorithm VIII and the best architecture found by sub-optimal cross validation (highlighted in bold on Table 4.5).

4.2.4 Conclusion

Throughout this Section we have presented a way to replace random forests with neural networks in the GENIE3 method and analysed the results. We proved that with a near optimal architecture, neural networks could outperform random forests even though the problem setting should a priori favour random forests. We then proceeded to describe a method in order to find a decent architecture using a cross-validation method while maintaining decent computation time. Finally we showed that by training multiple networks and averaging their importances we could enhance the results.

We tested all of these methods, leading to the summary of the results (Table 4.8). As we can see, we are able to achieve near GENIE3 performances (except for set number 2 which for some reasons seems to be ill-suited for neural networks) by only testing 12 architectures. Thus it is likely that by increasing the number of possible architectures and the number of networks in the ensemble we could enhance performances even further. This would allow us to perform as well, if not better than random forests.

		Method used				
		First try	Set 1 optimal architecture	Sub-optimal CV	Ensemble + sub-optimal CV	GENIE3
Set 1	I.L.	0.1396	0.075	0.1353	0.1407	**0.152**
	B.P. + I.L.	0.1064	**0.1575**	0.1392	0.1419	
Set 2	I.L.	0.099	0.0923	0.1177	**0.1182**	**0.154**
	B.P. + I.L.	0.0977	0.1172	0.1041	0.1072	
Set 3	I.L.	0.1813	0.1656	0.2014	0.2011	**0.228**
	B.P. + I.L.	0.1939	0.1952	0.2168	**0.2174**	
Set 4	I.L.	0.1927	0.137	0.198	0.1997	**0.209**
	B.P. + I.L.	0.205	0.1741	0.2046	**0.206**	
Set 5	I.L.	0.1692	0.0832	0.1715	0.1793	**0.195**
	B.P. + I.L.	0.1734	0.1553	0.1933	**0.1963**	

Table 4.8: Summary of the results obtained for the 5 DREAM4 datasets, showing the performance improvement as shown throughout Section 4.2.

4.3 Multi-output approach

This Section aims at defining a method to use the B.P. method in order to build a single network to compute the full importances matrix. To this end, we split this Section in the following Subsections.

1. **Motivation.** In which we motivate the multi-output approach.
2. **Network architecture.** In which we describe the architecture type that will be used for multi-output regression.
3. **Change in the backpropagation technique.** Here we give an initialisation method for the B.P. which allows us to compute importances for each output separately.
4. **Discussion on the network architecture.** As in Section 4.2, we will look at a multitude of different architectures and look at their effect on the AUPR.
5. **The cross validation problem.** In which we explain why the cross validation method proposed in Section 4.2 might not work in the multi-output setting.
6. **Multi-output conclusion.**

4.3.1 Motivation

First, the idea of using a multi-output network is well motivated biologically. In fact, transcription factors are shared between genes (i.e. genes work per modules). This means that a group of genes g might influence multiple genes. This would thus be more easily detected by a multi-output network. Intuitively, the network would have an easier time selecting groups of inputs that act on groups of outputs if it is shown their contributions on all other genes simultaneously rather than one at a time.

Second, using this technique we are able to carry out the feature selection using a single network. This is a huge advantage in terms of computation times.

4.3.2 Network architecture

Let us describe the general architecture we will use in this Section. There are three main differences between the multi-output and single-output architecture.

- First, there are multiple outputs, thus the final layer is composed of 100 neurons where each output corresponds to a given gene expression.

- Second, we remove the invisible layer since we now want to use feature selection per output to fill the importances matrix. Note that there is no more regularisation parameters to be optimised.

- Finally, we introduce a smaller layer in the middle of the hidden layers. We will call this an "encoder layer" which is used to encode the features created by the first hidden layers of the network. This way the network is not able to build a "direct path" between $input_i$ and $output_i$ since we have the relation $input_i = output_i$ due to the method principle.

There are thus 3 architecture parameters to tune in the network (which can be seen on Figure 4.4). First we need to choose k which represents the number of layers before and after the encoder layer and n the number of neurons in these layers. Finally, we need to set l which is the number of neurons used in the encoder layer, thus we should have $l < n$. The effect of these parameters on the feature selection performance will be discussed in Subsection 4.3.4.

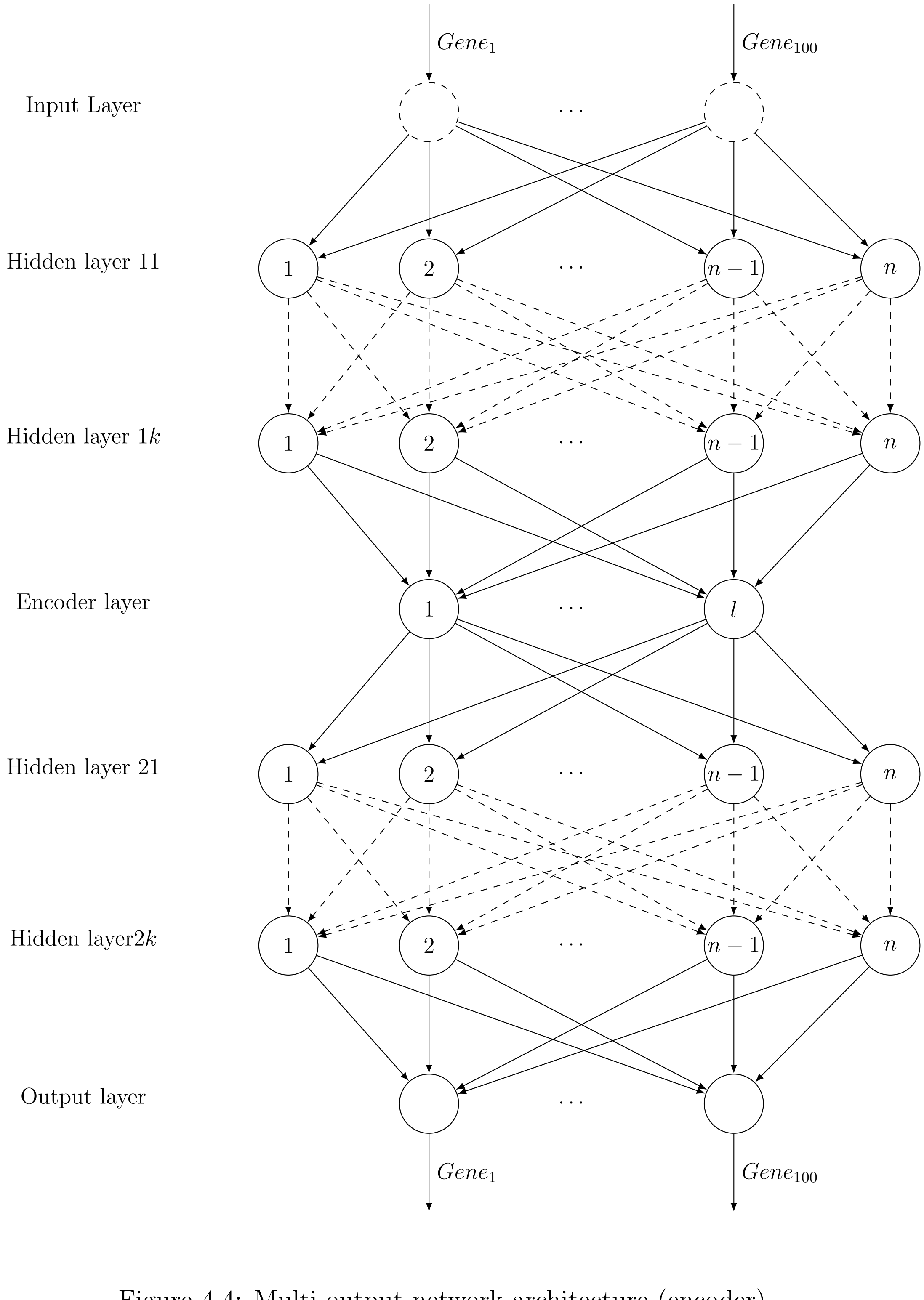

Figure 4.4: Multi-output network architecture (encoder).

4.3.3 Change in the backpropagation technique

As already mentioned, the big difference with the methods previously used is that we now want to carry out feature selection per output within the same neural network. To do this we use Algorithm IX. This way we are able to compute the feature importances of inputs for a given output.

Algorithm IX. *A way of filling the importances matrix using multi-output regression network.*

1. For i in [1,...,100]:
   (a) Initialise B.P. algorithm by setting importances of output neurons in [1,...,i-1,i+1,...,100] to 0
   (b) Compute feature importances using B.P. algorithm (As described in Subsection 3.2.1)
   (c) Set feature importance of input i to 0 (a gene can't be used to predict itself)
   (d) Vector obtained corresponds to the column i of importances matrix

4.3.4 Discussion on the network architecture

We will now discuss the results obtained by using the architecture described in Subsection 4.3.2 and the method described in Subsection 4.3.3. Similarly as for the single output approach, we decided to test a multitude of parameters combinations in order to see if some of them could be viable and could compare to the single output techniques. Table 4.9 summarises the combinations of parameters that were tested on set 1. As a reminder, we get an AUPR of 0.154 using the GENIE3 method on this same set.

Number of hidden layers before and after encoder layer (k)	Number of neurons per hidden layer (n)	Encoder layer size (l)
1	50	10
2	100	20
		30
		50
		100

Table 4.9: Possibilities for each of the three multi-output architecture parameters to optimise.

This has led us to test $2 * 2 * 5 = 20$ different architectures for which the AUPR are shown in Table 4.10. Unfortunately we are unable to achieve decent results and are far from single output methods' performances.

		k			
		1		2	
		n		n	
		50	100	50	100
l	10	**0.060**	0.036	0.031	0.039
	20	0.054	0.049	0.044	0.046
	30	0.047	0.058	0.046	0.045
	50	0.046	0.049	0.051	0.059
	100	0.058	0.049	0.049	0.048

Table 4.10: AUPR using different multi-output architecture (computed with Algorithm IX).

By taking a closer look at the importances matrix, we noticed that some columns had very high values compared to others, we thus decided to try and normalise the columns such that each has a 0 mean and 1 standard deviation. We then compute the AUPR on the matrix obtained and show the results on Table 4.11. As we can see, we are able to enhance a bit the results using this technique, unfortunately it is still insufficient to near single output performances. Furthermore, a new type of problem we didn't have before appears and will be discussed in next Subsection.

		k			
		1		2	
		n		n	
		50	100	50	100
l	10	0.067	0.032	0.029	0.050
	20	0.060	**0.068**	0.043	0.056
	30	0.059	0.067	0.039	0.046
	50	0.038	0.052	0.054	0.061
	100	0.056	0.048	0.040	0.037

Table 4.11: AUPR using different multi-output architecture and by normalising the importances matrix (computed with Algorithm IX).

As further research, two ideas can be looked at. First, a L1 penalty could be added at the output of the encoder layer, such that the selection of as few as possible of the outputs is enforced. Secondly, an intelligent training technique could be designed. Indeed, one could try to train the network for each output separately with only the other genes as input which could be done with some sort of "intelligent drop-out".

4.3.5 The cross validation problem

Even though we are not able to obtain decent results with multi-output architectures, we still need to discuss another problem that this method brings. Indeed, as stated before, it is impossible to know a priori the best architecture for the neural network. For the single output case, we have proposed a way to find a decent architecture nonetheless. To this end, we used the networks' MSE and considered that the most accurate network would also have the best AUPR. Unfortunately in the multi-output case, this does not hold true anymore. Indeed, the optimal structure in terms of MSE would be a bigger network where "direct paths" between $input_i$ and $output_i$ would have been learned. Although this would lead to low performance feature selection since this link can't exist due to the nature of the problem. Thus the technique described in Subsection 4.2.3 is unfortunately unusable in this particular case (at least for optimising the encoding layer).

4.3.6 Multi-output discussion

In this Section we gave a way to use multi-output neural networks to tackle the DREAM4 problem. We were able to obtain better than "random guessing" results using our B.P. algorithm. Indeed, in average, random guessing leads to an AUPR of 0.02 and our multi-output approach gives an AUPR of 0.068. This allowed us to give an example of the algorithm's modulability by proving that we are able to compute feature importances for a particular output while the network itself is trained for multiple ones. However, the performances were not as good as for single output in the particular case of the DREAM4 challenge. This may be due to the importances value varying too much from one gene to another, thus leading to poor performances. Finally we also noticed that in this particular setting, it would be very difficult to find an automatic way of tuning the network's architecture. Overall, the only reason to use this technique over single output networks is the computation time. Indeed, only one network has to be trained in this case which is a huge time gain compared to single output methods.

4.4 Conclusion

In this Chapter we managed to prove that the technique presented in this Chapter 3 could be used in order to tackle real challenges. We showed decent result against state-of-the-art methods and proved that with more refined tuning it could be possible to achieve similar (if not better) results. This is quite promising since the datasets on which the tests were carried out should a priori disfavour neural network due to the sheer lack of data samples. Furthermore we highlighted the two main drawbacks of using neural networks for feature selection :

- The huge variance in the performance due to the neural network architecture.
- The computation time required to train such models.

The first drawback is inexistent for random forests, and the second one is much less of a problem. However, we proposed methods in order to address these issues. Indeed, we showed that choosing the network with the best accuracy would often lead to better AUPR thus in order to find a near-optimal neural network architecture, cross-validation techniques can be used. Unfortunately, this would further increase the computation time required at the benefit of better results. We tried to tackle the computation time problem by training a single network with multiple outputs, unfortunately as shown in Section 4.3 the results were much worse than the single output method even though the computation time was much more acceptable.

On a side note, an interesting idea would be to have a look at the knock-out and knock-down datasets. Knockout and knockdown data are matrices in which the expression of all genes is measured, as the expression of each gene in turn is eliminated (for knock- outs) or substantially reduced (for knockdowns). Therefore, these give a priori information about the importance of the inputs. It would thus be a great way to test the possible extension suggested in Section 3.1 to further penalise high weights in the invisible layer for a priori non-informative features.

Overall we showed that using neural networks with our methods could give decent results and near (or probably surpass) state-of-the-art performances on problem with a very lowinput features over number of data samples ratio. Although it currently requires much more work, and much higher computation time to achieve similar performances, we are optimistic that on larger datasets it would in fact be the contrary. Indeed, in Chapter 3 we showed that in artificial settings with more samples (still reasonable amounts though), neural networks could actually outperform random forests.

Chapter 5

Conclusion and perspectives

In this thesis we took some inspirations from existing algorithms in order to propose new ones for selecting features using deep neural networks. More precisely we first showed that introducing a "one-to-one" connected layer we named "invisible layer" (I.L.) could allow to regularise the network easily using known methods such as LASSO and elastic-net regularisation (as proposed by [Li et al., 2015]). We then proposed a new "per neuron" measure of importance based on neural activation which is computed by back-propagation and averaged over all training samples (B.P.). We proceeded to mix the I.L. technique and the B.P. to achieve better results, leading to what we called the mixed method (B.P. + I.L.). Finally we tried to use a feature swapping technique (F.S.), which is a well-known technique that works with any machine learning algorithm.

The algorithms were first tested on three artificial datasets with multiple parameters. Thus, we obtained results on classification and regression problems with different level of noise and different neural architectures. This allowed us to compare each algorithm on multiple fronts: performances, computation time and constraints. We saw that the B.P. + I.L. algorithm was the most robust overall and almost always gave the best performances. However, this algorithm has the constraint that it can't be used as such on pre-trained networks. Indeed, the I.L. requires a precise neural architecture that needs to be trained. The results obtained were really promising since we managed to achieve our goal by surpassing (or equaling) random forests performances in every case (which was set to be our "state-of-the-art" comparison).

Seeing the promising results obtained on the artificial datasets we decided to tackle the DREAM4 challenge, which was supposedly an ill-suited problem for neural networks due to the very small number of samples available in the datasets. To this end, we used the mixed method (I.L. + B.P.) and compared our results to those of the GENIE3 method [Huynh-Thu et al., 2010] (which gives state-of-the-art results). We found that with the optimal architecture, neural networks could surpass GENIE3 performances, however this architecture cannot be known a priori. We thus provided a general method based on cross-validation that allows to automatically find a near optimal architecture. Using this method we managed to achieve near GENIE3 performances. However since only a few different architectures were tested, we believe that by automatically testing more architectures (leading to increased computation times) better ones would be found and we would equal or even surpass GENIE3 results at the cost of heavy computation times.

In Chapter 3, we gave insight on how the proposed algorithms could also be modified in order to carry out other tasks. For example, we proposed another formula that could be used with our back-propagation method in order to prune networks without losing accuracy which should be looked at in futur works. Furthermore, even though we used the same algorithms for all of the datasets to prove their generality, we provided hints on how they could be enhanced case by case. Among others, we gave an alternative initialisation method for the back-propagation algorithm in a binary classification setting (see Equation 3.7).

Let us give the perspectives of this work and improvements that could be made. First, concerning the DREAM4 challenge, as already mentioned we should try a wider variety of architectures in order to find one that is closer to the optimum, this would lead to a non-marginal boost in performances. Second, other initialisation techniques should be tried for the back-propagation with classification datasets to boost performances. Third, convolutional and recurrent neural networks should be looked at. The back-propagation formula should probably be modified to tackle those particular networks by associating importances to group of neurons rather than "per neuron" to take into account the neural structure. Finally, other regularisation techniques could be tried to introduce a priori information in the network.

As a last word, what we have proposed here are very modulable algorithms. We want to stress that the main ideas used here work very well, i.e. introducing a one-to-one layer ([Li et al., 2015]) and computing a measure based on neural activation ([Debaditya et al., 2015]) by using back-propagation ([Kim et al., 2016]). We proposed an original algorithm which merges all of the ideas stated above. In order to combine them, we created new formulae that work quite well for many applications but that can be enhanced depending on the problem. An example of this is the way we computed the feature importances for each output one at a time in Section 4.3. All in all, the algorithms we gave are quite promising and fare well against random forests which are often used in bioinformatics to give state-of-the-art performances on feature selection tasks.

Appendix A

Hypercube dataset generation

Here, we define the generation of our artificial binary classification problem. It is generated the same way as the Madelon problem (which was part of a challenge, for which the results are given in [Guyon et al., 2004]), i.e. it consists of an hypercube of dimension n where each vertices' neighbourhood is attributed one of two classes.

The generation creates clusters of points normally distributed (std=1) about vertices of a 2-sided hypercube, and assigns an equal number of clusters to each class. Each input thus corresponds to the coordinate one of the n axis. It also introduces interdependence between these features and adds various types of further noise to the data. The amount of noisy variables added is a parameter. The generation is the one implemented in scikit-learn except a small modification that had to be made in order to know beforehand the relevant features. Further information can be obtained in [skl, b].

In our case we decided to use an hypercube of dimension 25 for all of our tests. The noise added varies according to the test made.

Appendix B

Hardware and software details

It is worth detailing briefly the hardware and software tools used throughout this thesis.

First, let us talk about the computer on which the tests were run. The computer runs linux debian jessie and has the following characteristics:

- 64GB of RAM
- CPU : INTEL6850K
- GPU : NVIDIA-GTX1080 (2560 CUDA cores, 8GB Vram)

The most important part in our case was the GPU. All the computations made in neural networks can be heavily parallelised, running the software on the GPU leads to a huge performance increase.

For the software, we used Tensorflow (v0.7) as it is a very low-level package that allows extremely customised networks. Even though it often requires more work to build simple networks, it is useful to implement more in-depth networks. In our case, it came in very handy for the back-propagation method. We first tried to get the parameters from the networks and run the B.P. algorithm on CPU. Even though our CPU is quite powerful, running the back-propagation algorithm took a long time (approximately 20 minutes to compute the feature importances for a 4 hidden layers network on 2000 training samples). However, Tensorflow provides a full set of matrices and vectors operations which can be run on GPU. By coding our formula in a smart way (i.e. only with matrices operations, such that it can be parallelised by Tensorflow) we were able to compute the importances in less than 30 seconds. Overall, even though it is easier to use other ANN libraries to train neural networks for practical applications (often, these libraries are a simple abstraction of lower-level ones), Tensorflow proved to be a great tool for us to implement our techniques.

Bibliography

[skl, a] Scikit-learn decision tree classifier. `http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html`.

[skl, b] Scikit-learn hypercube classification. `http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html`.

[skl, c] Scikit-learn hypercube classificationregression. `http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html`.

[Altmann et al., 2010] Altmann, A., Toloşi, L., Sander, O., and Lengauer, T. (2010). Permutation importance: a corrected feature importance measure. *Bioinformatics*, 26(10):1340–1347.

[Chen et al., 2007] Chen, X., Liu, C.-T., Zhang, M., and Zhang, H. (2007). A forest-based approach to identifying gene and gene–gene interactions. *Proceedings of the National Academy of Sciences*, 104(49):19199–19203.

[Debaditya et al., 2015] Debaditya, R., Sri, R. M. K., and Krishna, M. C. (2015). Feature selection using deep neural networks.

[Guyon and Elisseeff, 2003] Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182.

[Guyon et al., 2004] Guyon, I., Gunn, S. R., Ben-Hur, A., and Dror, G. (2004). Result analysis of the nips 2003 feature selection challenge. In *NIPS*, volume 4, pages 545–552.

[Guyon et al., 2002] Guyon, I., Weston, J., Barnhill, S., and Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1):389–422.

[Huynh-Thu et al., 2010] Huynh-Thu, V. A., Irrthum, A., Wehenkel, L., and Geurts, P. (2010). Inferring regulatory networks from expression data using tree-based methods. *PLOS ONE.*

[Kim et al., 2016] Kim, S. G., Theera-Ampornpunt, N., Fang, C.-H., Harwani, M., Grama, A., and Chaterji, S. (2016). Opening up the blackbox: an interpretable deep neural network-based classifier for cell-type specific enhancer predictions. *BMC Systems Biology*, 10(2):244–258.

[Kingma and Ba, 2014] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Leray and Gallinari, 1999] Leray, P. and Gallinari, P. (1999). Feature selection with neural networks.

[Li et al., 2015] Li, Y., yu Chen, C., and Wasserman, W. W. (2015). Deep feature selection: Theory and application to identify enhancers and promoters. *JOURNAL OF COMPUTATIONAL BIOLOGY*, pages 1–15.

[Marbach et al., 2009] Marbach, D., Schaffter, T., Mattiussi, C., and Floreano, D. (2009). Generating realistic in silico gene networks for performance assessment of reverse engineering methods. *Journal of Computational Biology*, 16(2):229–239.

[Montavon et al., 2017] Montavon, G., Lapuschkin, S., Binder, A., Samek, W., and Müller, K.-R. (2017). Explaining nonlinear classification decisions with deep taylor decomposition. *Pattern Recognition*, 65:211–222.

[Nair and Hinton, 2010] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814.

[Nilsson et al., 2007] Nilsson, R., Peña, J. M., Björkegren, J., and Tegnér, J. (2007). Consistent feature selection for pattern recognition in polynomial time. *Journal of Machine Learning Research*, 8(Mar):589–612.

[Qi, 2012] Qi, Y. (2012). Random forest for bioinformatics. In *Ensemble machine learning*, pages 307–323. Springer.

[Rumelhart et al., 1988] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

[Saeys et al., 2007] Saeys, Y., Inza, I., and Larraaga, P. (2007). A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23(19):2507.

[Shrikumar et al., 2017] Shrikumar, A., Greenside, P., and Kundaje, A. (2017). Learning important features through propagating activation differences. *arXiv preprint arXiv:1704.02685*.

[Srivastava et al., 2014] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

[Stolovitzk et al., 2007] Stolovitzk, G., Monroe, D., and Califano, A. (2007). Dialogue on reverse- engineering assessment and methods: The dream of high-throughput pathway inference. *Annals of the New York Academy of Sciences*, 1115:11–22.

[Stolovitzky et al., 2009] Stolovitzky, G., Prill, R., and Califano, A. (2009). Lessons from the dream2 challenges. *Annals of the New York Academy of Sciences*, 1158:159–95.

[Strobl et al., 2008] Strobl, C., Boulesteix, A.-L., Kneib, T., Augustin, T., and Zeileis, A. (2008). Conditional variable importance for random forests. *BMC bioinformatics*, 9(1):307.

[Wang et al., 2010] Wang, M., Chen, X., and Zhang, H. (2010). Maximal conditional chi-square importance in random forests. *Bioinformatics*, 26(6):831–837.

[Zou and Hastie, 2005] Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320.