GENE EXPRESSION PREDICTION BASED ON DEEP LEARNING

A Thesis presented to
the Faculty of the Graduate School
at the University of Missouri

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by

RUI XIE
Dr. JIANLIN CHENG, Thesis Supervisor

MAY 2016

The undersigned, appointed by the Dean of the Graduate School, have examined the dissertation entitled:

GENE EXPRESSION PREDICTION

BASED ON DEEP LEARNING

presented by Rui Xie,

a candidate for the degree of Master of Science and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Jianlin Cheng

Dr. Dong Xu

Dr. Yunxin Zhao

Dr. Zhihai He

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to my advisor Prof. Jianlin Cheng, for his excellent guidance, caring and patience. I attribute the level of my masters degree to his encouragement and effort. This project and my graduate study would not have been completed without his support.

Secondly, I would like to thank Prof. Xinhua Shi for providing me with excellent guidance and material to assit me with my work on the Deep Learning model. I appreciate her professional suggestions for my project.

My sincere thanks also goes to my committee members, Prof. Dong Xu, Prof. Yunxin Zhao and Prof. Zhihai He, for reviewing and giving suggestions regarding my thesis and for the encouragement they gave when I needed help.

Special thanks goes to Jie Hou, Jilong Li, Haiou Li and Renzhi Cao for offering lots of help with my thesis research. A lot of data processing and model tuning were made under their guidances and suggestions. Finally, I would like to thank my family and friends for their consistent love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Gene expression is a critical process in a biological system that is influenced and modulated by many factors including genetic variation. Thus, it is important to understand how genotypes affect the gene expression levels. Although several approaches have been implemented, we proposed a deep learning regression model to learn complex feature representation and to deal with over-fitting. In our experiment, the deep learning model produced results that are comparable to results generated by other methods by applying an independent test data set.

This thesis has several contributions. First, we propose an accurate predicting model based on deep learning to extract useful features with multilayer perceptron and stacked denoising auto-encoders after preprocessing the input data. Second, we ran a test on an independent dataset for several approaches to evaluate the performance of a multilayer perceptron with stacked denoising auto-encoders. Third, we further improved our model by adding a dropout technique to prevent overfitting. The result shows that dropout improved the model when we compared the result of our model with results of other existing approaches to evaluate its performance with a test data set. Finally, we present a software package that allows users to train the model with their own data and make predictions. An instruction on how to use this software package was also provided.

Chapter 1

Introduction

In this chapter, we will provide an overview of the gene expression prediction problem to be explored in this thesis. In addition, we will discuss the existing approaches. Finally, we will give an outline for every chapter of this thesis.

1.1 Introduction to Gene Expression Prediction

In a biological system, many factors such as genetic variation are influencing and modulating gene expression. Meanwhile, genetic variation is reflecting the genetic differences among individuals in human population. Such variation can be at different levels, ranging from single nucleotide polymorphisms (SNPs) to structural variation including copy number variants (CNVs). In addition, studies have shown that many SNPs are significantly associated with the expression of various genes, although not many predictive models are offered in these studies, making it hard for an individual to predict their own gene expression from their genotype.

Building an accurate predictive model can help us assess the effect of genetic variations on gene expression, which can improve our understanding of how genetic variation leads to gene expression variation: for example, by making contributions to human health and disease.

With its pass in many areas, deep learning is a possible solution for solving bioinformatics problems. This possibility has led us to explore the possibility to learn a deep learning predictive model of gene expression, which takes the combination of different genotypes and predicts the expression of a gene in individuals given their genotype value.

1.2 Existing Approaches and Challenges

In this section, we will discuss the exsiting approaches for gene expression prediction and challenge we are going to address. These apporoaches have given us a guidelines and provided benchmarks to evaluate our methods.

There are two popular approaches for gene expression prediction and we used those methods for our analyses and compared their performances.

Meanwhile, we are facing several challenges that need to be solved.

1.2.1 Lasso

The first method to consider is Lasso [1, 2, 3], which is a shrinkage and selection method for the linear regression method [4]. It minimizes the usual sum of squared errors, with a bound on the sum of the absolute values of the coefficients.

Lasso is useful in some situation because of its tendency to prefer solutions with fewer parameter values. Thus, it effectively reduces the number of variables upon which the given solution is dependent. For this reason, the Lasso and its variants are fundamental to the field of compressed sensing. Under certain conditions, it can recover the exact set of non-zero weights.

Mathematically, Lasso consists of a linear model trained with ℓ_1 prior as regularizer. The objective function [5] is a minimization problem, which can be expressed as Equation 1.1:

$$min \frac{1}{2n_{samples}} ||X\omega - y||_2^2 + \alpha||\omega||_1 \tag{1.1}$$

where α is a constant and $||w||_1$ is the ℓ_1-norm of the parameter vector. Thus the Lasso estimate solves the minimization of the least-squares penalty with $\alpha||w||_1$ added. If α is large enough, some coefficients will be shrunk to an exact zero. Therefore Lasso simutaneously produces both an accurate and sparse model, which makes it a feasible variable selection method.

1.2.2 Random Forests

The second solution is Random Forests[4, 6, 7, 8], which is an ensemble learning method for classification, regression and other tasks. By constructing a multitude of decision trees at training time, Random Forests is capable of performing classification or mean prediction (regression) of the individual trees. In addition, Random Forests corrects for decision trees' habit of overfitting to their training set[9].

The Random Forests algorithm [10] (for both classification and regression) is as follows:

- From the original data, draw n bootstrap samples.
- For each of the bootstrap samples, grow an unpruned classification or regression tree, with the following modification: At each node, rather than choosing the best split among all predictors, randomly sample m of the predictors and choose the best split from among those variables (Bagging can be thought of as the special case of Random Forests obtained when $m = p$, the number of predictors).
- Predict new data by aggregating the predictions of the n trees (i.e., majority votes for classification, average for regression).

An estimate of the error rate can be obtained, based on the training data, by the following:

- At each bootstrap iteration, predict the data not in the bootstrap sample using the tree grown with the bootstrap sample.
- Aggregate the predictions.

Here we also use Figure 1.1 to illustrate an example of the Random Forests regression model.

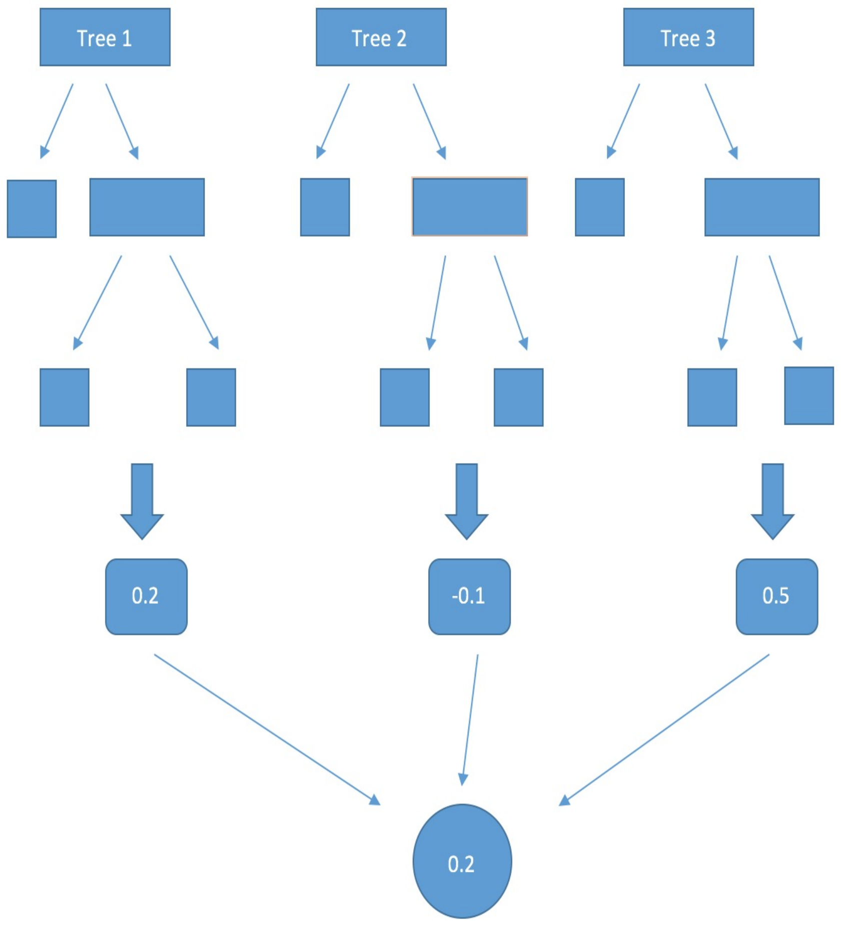


Figure 1.1: An example of the Random Forest Regression Model

1.2.3 Challenges

This thesis introduces a regression model based on Multilayer Perceptron (MLP)and Stacked AutoEncoder (SAE). Given training features, users are able to predict gene expression value. It is essential that the system perform precise gene expression value prediction. This is not an easy task due to the following reasons.

First, sample features contain missing values, which needs an appropriate preprocessing to minimize the negative effect (noise).

Second, gene expression prediction is computational expensive since we are dealing with samples that have thousands of features and are trying to predict thousands of outcomes for every given sample.

Third, we need to handle overfitting, which is a common problem in the field of machine learning.

1.3 Thesis Outline and Contributions

The remainder of this thesis is structured as follows.

In chapter 2, we begin to introduce the data for our model. Then we introduce a prediction model based on deep learning. Specifically, we propose a model of multilayer perceptrons and auto-encoders to consistently train with given features and optimize through back propagation. The first step is based on training an auto encoder with a stochastic gradient descent algorithm and the second step is using the two auto encoders as two hidden layers and train with back propagation algorithm. Additionally, we use cross validation to select an optimal model for all three prediction methods. When the training is finished, the results are evaluated on the independent test data set.

In Chapter 3, we further improve the MLP with auto-encoder by adopting the dropout technique. This technique has proven to prevent over fitting and bring more stable results.

In Chapter 4, we will summarize the results and discuss the future work and limitations.

Finally, in Appendix A, we provide a brief overview of the tool used for solving this problem, including scikit-learn and pylearn2. We will also provide the script of data pre-processing and the configuration of YAML files for building the deep learning architecture. Additionally, the next section will provide guidance of input, output file and how to run the software package with Linux command.

To summarize, the contributions of this thesis are two-fold: (1)to solve the gene expression prediction with less error and stable training and (2) to provide the scientific community with access to fast, reliable and freely available prediction software to facilitate gene expression research.

Chapter 2

Prediction Model Based on Deep Learning

2.1 Abstract

In this chapter, we will first introduce the relationship between genotype and gene expression as well as the pre-processing of our training data set.

Second, we will introduce a prediction model based on deep learning as well as the Multilayer Perceptron and Stacked Denoising Auto-Encoder. The model starts training with given features and optimizing through a backpropagation algorithm. The first step is based on training the auto encoder with a stochastic gradient descent algorithm and the second step is using the two auto encoders as two hidden layers and train with multilayer perceptron. A backpropagation algorithm is used for optimization. After training, we will evaluated the performance on an independent data set and compared the results with different methods including Lasso and Random

Forests.

2.2 Background

To better understand the gene expression prediction problem we are dealing with, we will first introduce the concept of genotype and gene expression.

2.2.1 Introduction to Genotypes

The Genotype is that part (DNA sequence) of the genetic makeup of a cell, and therefore of an organism or individual, which determines a specific characteristic (phenotype) of that cell/organism/individual. Genotype is one of three factors that determine phenotype, the other two being inherited epigenetic factors, and non-inherited environmental factors. DNA mutations which are acquired rather than inherited, such as cancer mutations, are not part of the individual's genotype. Hence, scientists and physicians sometimes talk for example about the (geno)type of a particular cancer, that is the genotype of the disease as distinct from the diseased.

2.2.2 Introduction to Phenotype

It is essential to distinguish the descriptors of the organism, its genotype and phenotype, from the material objects that are being described. The genotype is the descriptor of the genome which is the set of physical DNA molecules inherited from the organism's parents. The phenotype is the descriptor of the phenome, the manifest physical properties of the organism, its physiology, morphology and behavior[11].

Any given gene will usually cause an observable change in an organism, known as the phenotype. The terms genotype and phenotype are distinct for at least two reasons:

To distinguish the source of an observer's knowledge (one can know about genotype by observing DNA, one can know about phenotype by observing outward appearance of an organism).

Genotype and phenotype are not always directly correlated. Some genes only express a given phenotype in certain environmental conditions. Conversely, some phenotypes could be the result of multiple genotypes. Genotypes are often mistakenly referred to as phenotypes, which describe the end result of both the genetic and the environmental factors resulting in expressions related to genetic characterisitcs (e.g. blue eyes, hair color, or various hereditary diseases).

2.3 Introduction to Data Sets

In this section, we will introduce the background of predicting pheotypes from genotypes. In addition, a detail explaination describing the data set for our experiment is provided.

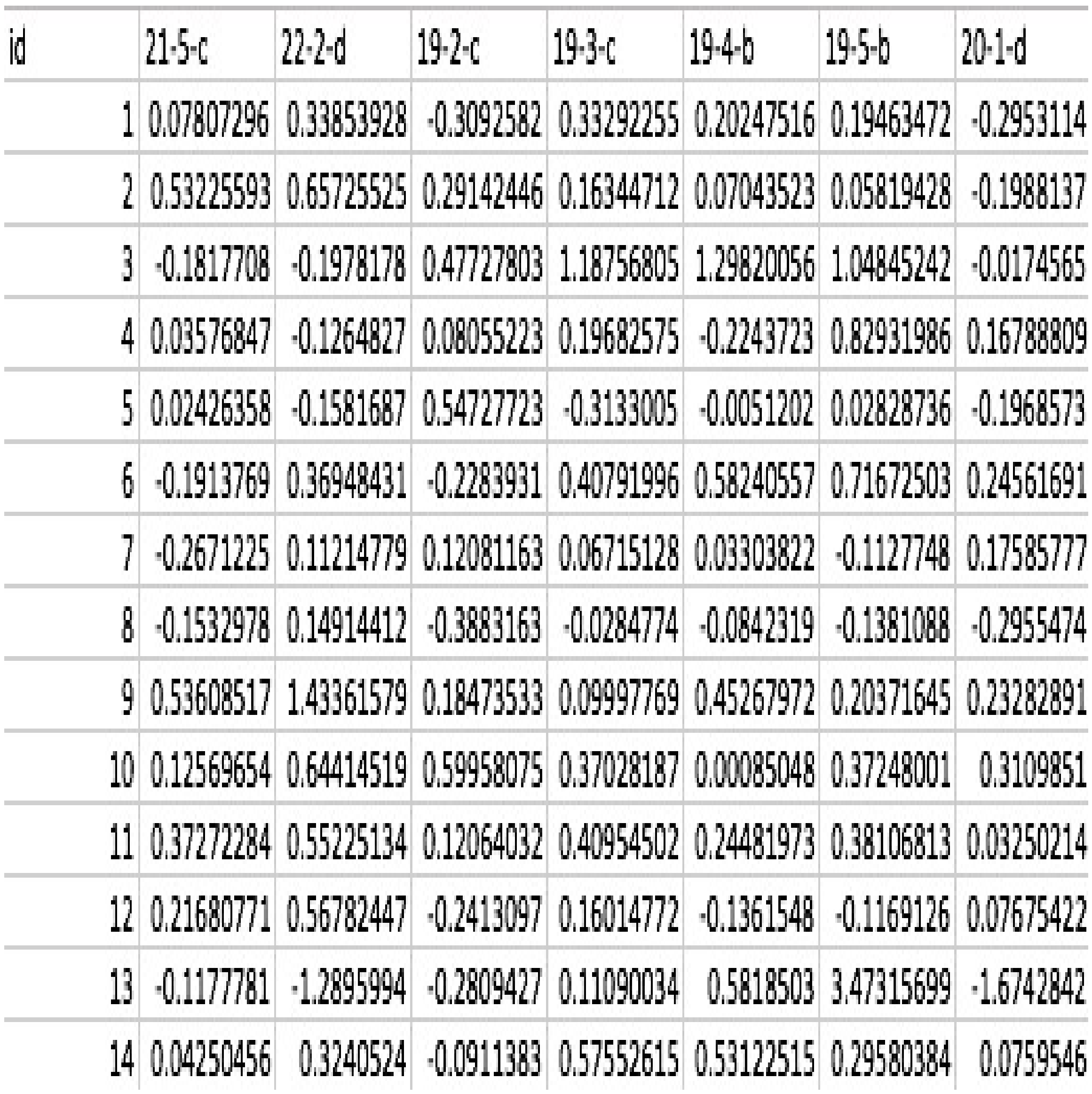
2.3.1 Introduction to Gene Expression

In genetics, gene expression is the most fundamental level at which the genotypes give rise to the phenotype, i.e., observable traits. Scientists are already using the new methods in gene expression analysis to extend the concept of the phenotype [12]. The genetic code stored in DNA is "interpreted" by gene expression, and the properties of the expression give rise to the organism's phenotype. Such phenotypes are often expressed by the synthesis of proteins that control the organism's shape, etc.

2.3.2 Data Set Preparation

For our deep learning model, our genotype input and gene expression data are came from yeast, which will be represented as matrices.

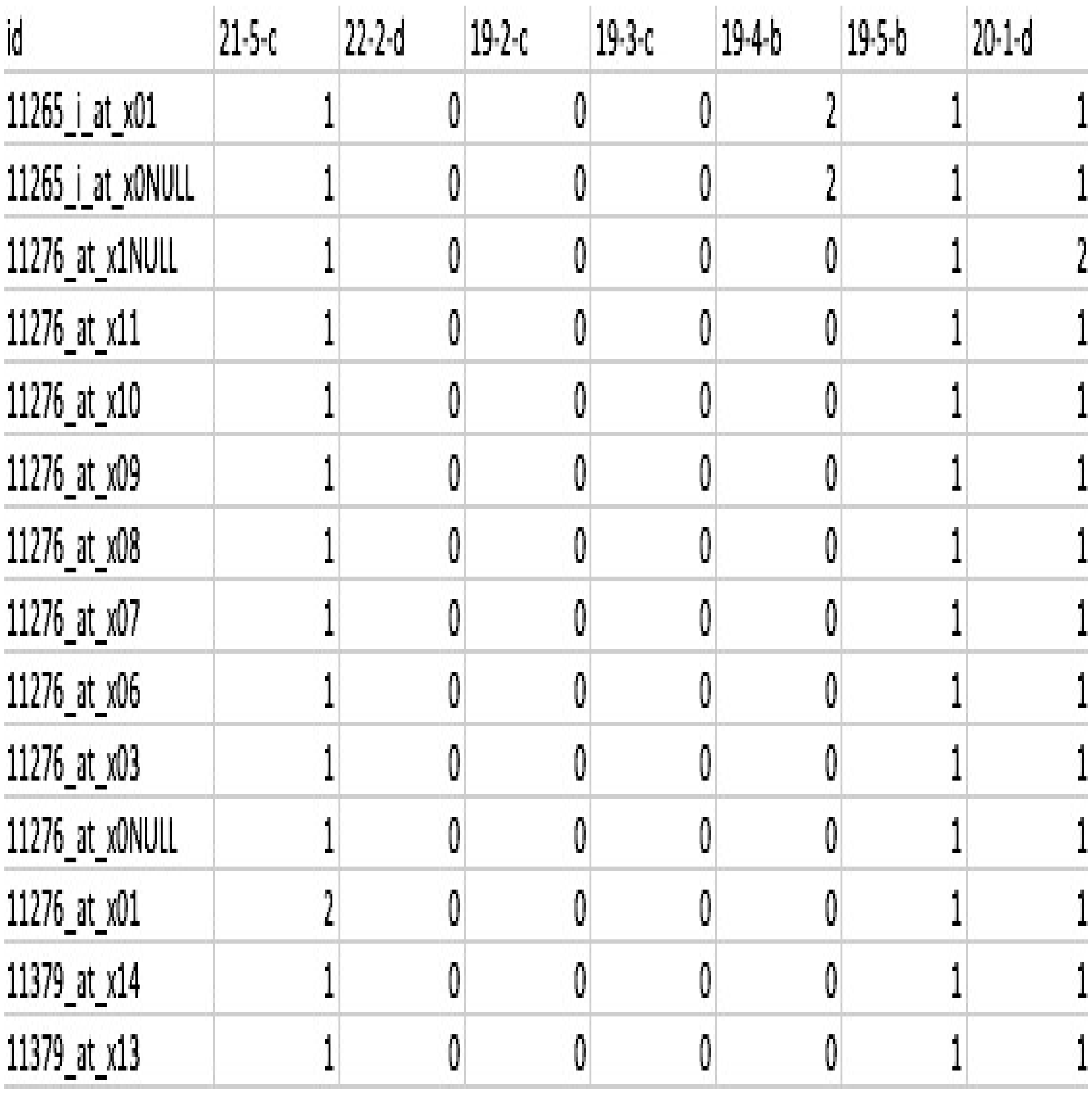
The expression matrix file holds values for gene expression. Each column is an individual yeast sample, and each row is an individual gene. Figure 2.1 illustrates the part of the gene expression data set.



id	21-5-c	22-2-d	19-2-c	19-3-c	19-4-b	19-5-b	20-1-d
1	0.07807296	0.33853928	-0.3092582	0.33292255	0.20247516	0.19463472	-0.2953114
2	0.53225593	0.65725525	0.29142446	0.16344712	0.07043523	0.05819428	-0.1988137
3	-0.1817708	-0.1978178	0.47727803	1.18756805	1.29820056	1.04845242	-0.0174565
4	0.03576847	-0.1264827	0.08055223	0.19682575	-0.2243723	0.82931986	0.16788809
5	0.02426358	-0.1581687	0.54727723	-0.3133005	-0.0051202	0.02828736	-0.1968573
6	-0.1913769	0.36948431	-0.2283931	0.40791996	0.58240557	0.71672503	0.24561691
7	-0.2671225	0.11214779	0.12081163	0.06715128	0.03303822	-0.1127748	0.17585777
8	-0.1532978	0.14914412	-0.3883163	-0.0284774	-0.0842319	-0.1381088	-0.2955474
9	0.53608517	1.43361579	0.18473533	0.09997769	0.45267972	0.20371645	0.23282891
10	0.12569654	0.64414519	0.59958075	0.37028187	0.00085048	0.37248001	0.3109851
11	0.37272284	0.55225134	0.12064032	0.40954502	0.24481973	0.38106813	0.03250214
12	0.21680771	0.56782447	-0.2413097	0.16014772	-0.1361548	-0.1169126	0.07675422
13	-0.1177781	-1.2895994	-0.2809427	0.11090034	0.5818503	3.47315699	-1.6742842
14	0.04250456	0.3240524	-0.0911383	0.57552615	0.53122515	0.29580384	0.0759546

Figure 2.1: Part of the gene expression data set
(A)Each column is an individual yeast sample, and each row represents a specific location in the yeast genome.
(B)The continuous value in each cell denote the gene expression value and NULL is missing data.

The genotype file is similar. Each column is an individual yeast sample, and each row represents a specific location in the yeast genome. The yeast that the data comes from are crosses, which means that they are a mix of two strains. So for any location a sample could have the genotype of either parent strain. In the genotype data file, 0 means it came from one parent strain, 1 means it came from the second parent strain and 2 is missing data. The Figure 2.2 shows part of the genotype value including sample names and specific positions:



id	21-5-c	22-2-d	19-2-c	19-3-c	19-4-b	19-5-b	20-1-d
11265_i_at_x01	1	0	0	0	2	1	1
11265_i_at_x0NULL	1	0	0	0	2	1	1
11276_at_x1NULL	1	0	0	0	0	1	2
11276_at_x11	1	0	0	0	0	1	1
11276_at_x10	1	0	0	0	0	1	1
11276_at_x09	1	0	0	0	0	1	1
11276_at_x08	1	0	0	0	0	1	1
11276_at_x07	1	0	0	0	0	1	1
11276_at_x06	1	0	0	0	0	1	1
11276_at_x03	1	0	0	0	0	1	1
11276_at_x0NULL	1	0	0	0	0	1	1
11276_at_x01	2	0	0	0	0	1	1
11379_at_x14	1	0	0	0	0	1	1
11379_at_x13	1	0	0	0	0	1	1

Figure 2.2: Part of Genotype Feature
(A)Each column is an individual yeast sample, and each row represents a specific location in the yeast genome.
(B)0 means it came from one parent strain, 1 means it came from the second parent strain and 2 is missing data.

For our experiment, we obtained 112 samples of genotype and gene expression of yeast. Later we split the dataset for cross validation.

The gene expression data consists of 7085 genes and the genotype data consists of 2956 specific locations. Since there are many missing values in the gene expression data, which provide no contribution to the prediction, we filtered hundreds of genes with 100 percent missing value in gene expressions data, the number of remaining effective gene expressions are 6611.

To process the raw data, we used the toolkit provided by Scikit-learn, referred to as Imputer and MinMaxScaler[13]. Both toolkits scale and translate each feature individually such that it is in the given range on the training set, i.e. between zero and one.

2.4 Deep Learning Regression Model

Since we are dealing with output of continuous values, we need to build up a regression model based on deep learning. The work flow of building a deep learning model is represented in Figure 2.3.

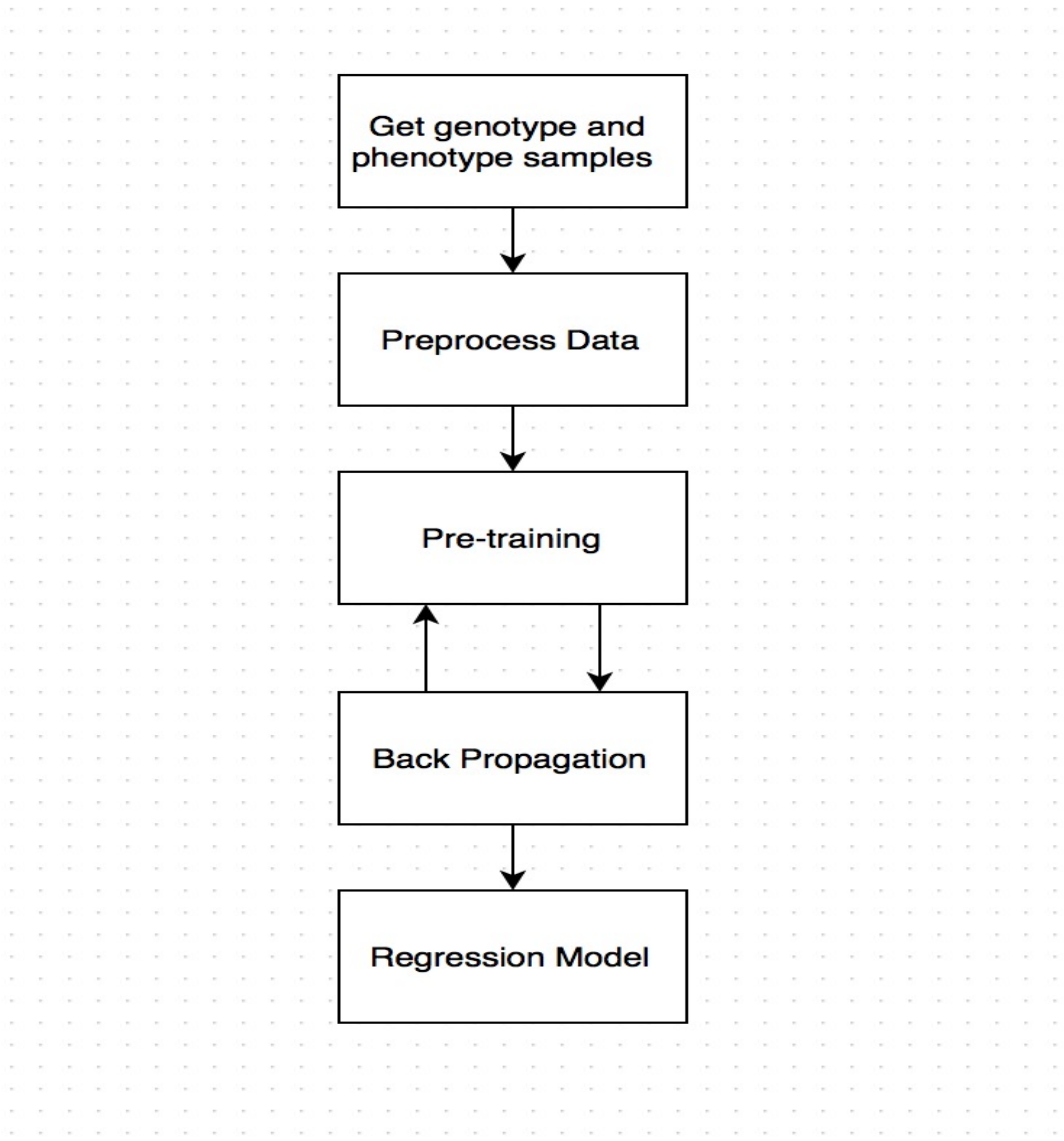


Figure 2.3: Flow Chart for Deep Learning Process

The model will first processes the raw input and then performs pre-training. When it reaches top layer, the model will finetune with backpropagation. The algorithm will stops when it reaches convergency.

2.4.1 Linear Regression

Statistically, linear regression is a model that represents the relationship between a variable Y and one or more variables X.

In our deep learning regression model, the linear regression of the final layer can be represented as Equation 2.1:

$$f(x) = \omega^T x + b \tag{2.1}$$

where the w matrix is the weight and b is the bias, and both are trained to minimize objective function.

2.4.2 Deep Neural Network

One of the most common description of deep neural network is that the a network has two or more layers of hidden processing neurons. In contrast, a neural network consisting of three layers is a fairly shallow network, which means the network computes the features using only one hidden layer. So a network with more than three layers can provide computation of much more complex features of the input. Thus, we use Figure 2.4 to illustrate the form of a shallow network:

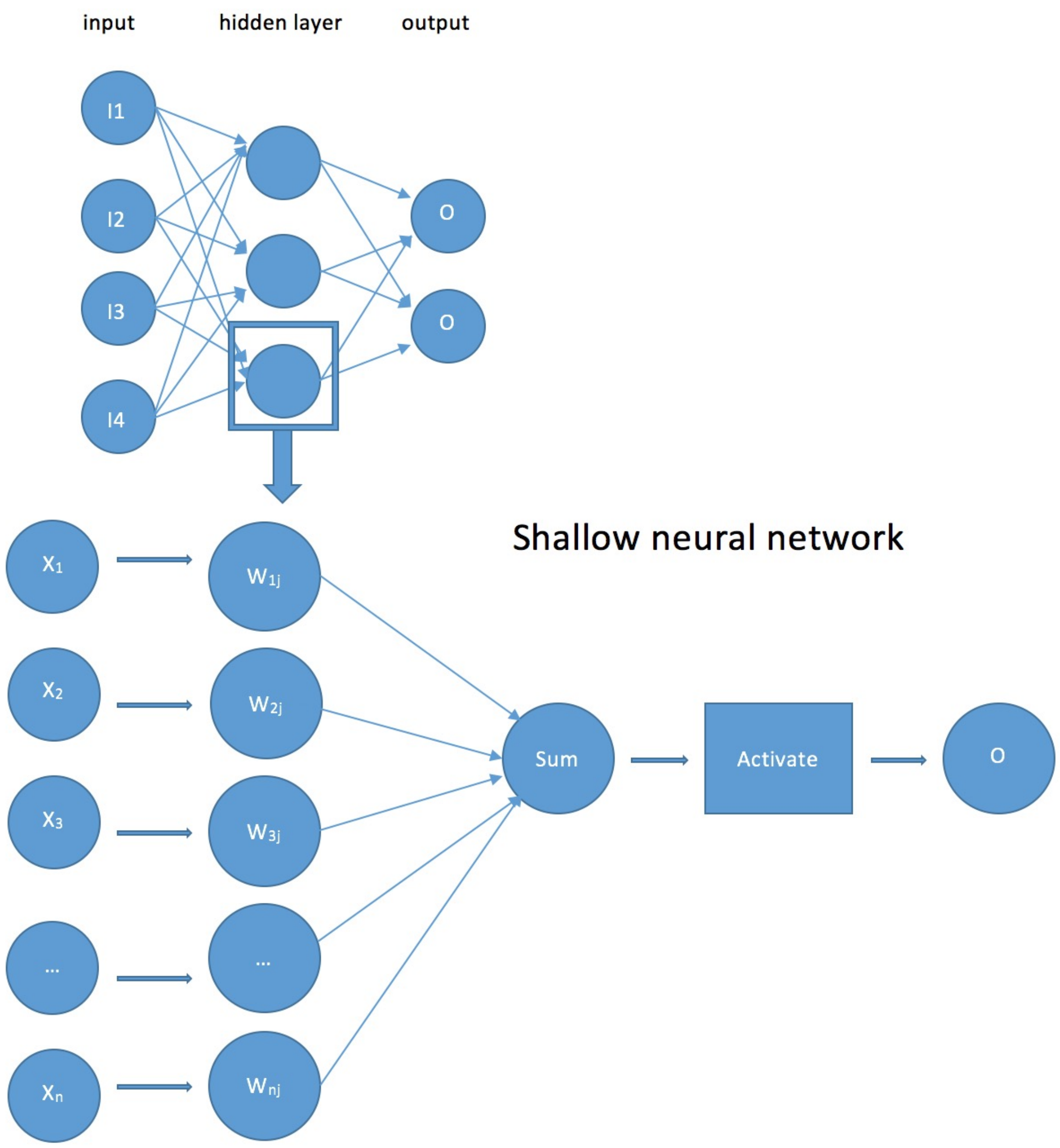


Figure 2.4: An example for shallow neural network model, which only consists of three layers.

2.5 Multilayer Perceptron

Multilayer perceptron is a feedfoward network that tries to map a set of input onto a set of output. A MLP has multiple layers of nodes while each layer is fully connected with the next one. Except for the input layer, each node of hidden layer is with a nonlinear activation function. Beside that, a backpropagation algorithm is used to train the network[14].

2.5.1 Activation Function

There are two main activation functions used for training. With ranges from -1 to 1, the hyperbolic tangent is used, which is described by Equation 2.2.

$$y(v_i) = tanh(v_i) \tag{2.2}$$

for ranges from 0 to 1, the logistic function is used, which is described by Equation 2.3

$$y(v_i) = (1 + e^{-v_i})^{-1} \tag{2.3}$$

Here y_i is the output of the ith node (neuron) and v_i is the weighted sum of the input synapses.

2.5.2 Learning through Backpropagation

The network is learning through changing connection weights after the data of each neuron in the layers is processed. With the amount of error in the output compared to the expected result, we were able to perform supervised learning. Here is a simple example:

We calculate the error e in node j in nth row (training sample) by Equation 2.4.

$$error_j(n) = d_j(n) - y_j(n) \tag{2.4}$$

where y is the target value and d is the expected value. After that, we will make the correction based on those corrections which minimize the error in the entire output, given by Equation 2.5.

$$\varepsilon(n) = \frac{1}{2}\sum_i error_j^2(n) \tag{2.5}$$

After gradient descent, the needed change in each weight represented as Equation 2.6.

$$\Delta\omega_{ji}(n) = -\eta\frac{\partial\varepsilon(n)}{\partial\upsilon_j(n)}y_i(n) \tag{2.6}$$

where the output of the previous neuron is represented by y and the learning rate is represented by η.

With the various induced local fields, the derivative can be calculated. The derivative for an output node can be represented as Equation 2.7.

$$-\frac{\partial\varepsilon(n)}{\partial\upsilon_j(n)} = error_j(n)\phi'(\upsilon_j(n)) \tag{2.7}$$

where ϕ' is the derivative of the activation function described above, which itself does not vary. The analysis is more difficult for the change in weights to a hidden node, but it can be shown that the relevant derivative is expressed by Equation 2.8.

$$-\frac{\partial \varepsilon(n)}{\partial \upsilon_j(n)} = \phi'(\upsilon_j(n)) \sum_k -\frac{\partial \varepsilon(n)}{\partial \upsilon_k(n)} \omega_{kj}(n) \tag{2.8}$$

This depends on the change in weights of the th nodes, which represent the output layer. So to change the hidden layer weights, we must first change the output layer weights according to the derivative of the activation function, and so this algorithm represents a backpropagation of the activation function.

2.6 Stacked Denoising Auto-encoder

An Auto-encoder [15] is a type of neural network that provides learning efficient codings. The main goal of an auto-encoder is to learn a compressed, distributed representation (encoding) for a set of data, typically for the purpose of dimensionality reduction. For each autoencoder, the network tries to reproduce the provided input data by using supervised learning, thus the backpropagation method will be a suitable method in the supervised training of multi-layer networks[16]. We use Figure 2.5 to illustrate the form of a stackded denoising auto-encoder:

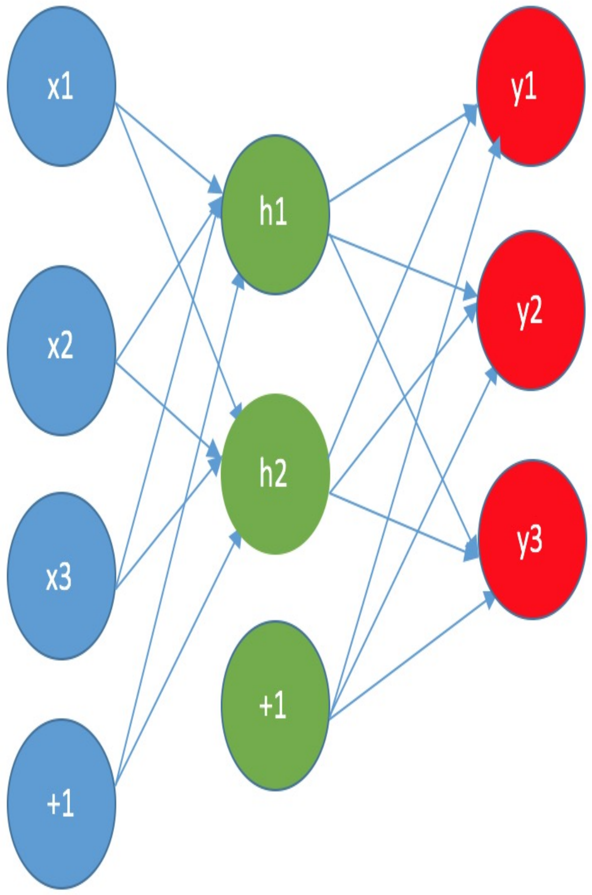


Figure 2.5: An example for Auto-encoder Model

2.6.1 Pre-training

Similar to Multilayer Perceptron (MLP), a simple form of the autoencoder is a feed-forward, non-recurrent neural net [17], with an input layer, an output layer and one or more hidden layers connecting them.

Although MLP is similar to a neural network, there is a difference between an MLP and an autoencoder. For an autoencoder, the output layer has equally many nodes as the input layer, and instead of training it to predict some target value y given inputs x, an autoencoder is trained to reconstruct its own inputs x and optimize through minimizing its objective function.

The training algorithm can be summarized as

- For each input x, do a feed-forward pass to compute the value of all nodes in the hidden layers after activation, then at the output layer to obtain an output $\widehat{x}$.
- Measure the deviation of $\widehat{x}$ from the input x (a common method is using squared error)
- Backpropagate the error through the net and perform weight updates (a common method is using the stochastic gradient descent algorithm).

In conclusion, the activations of the final hidden layer can be regarded as a compressed representation of the input if the hidden layers have fewer nodes than the input/output layers. In addition, activation functions that are commonly used in MLPs can be used in autoencoders. Moreover, the optimal solution to an auto-encoder is strongly related to principal component analysis (PCA) [18] if linear activations are used, or only a single sigmoid hidden layer [19].

An autoencoder can potentially learn the identity function and become useless, when the hidden layers are larger than the input layer. However, such autoencoders might still learn useful features according to some experimental results [20].

2.6.2 Corrupted Level

It is a common to add noise so that the data is shuffled around and a denoising auto-encoder can learn about that data by attempting to reconstruct it.

Recognizing the features within the noise that will allow it to classify the input is the main goal of the network. During the training of the network, a model is generated and a obejective function such as squared error measures the distance between that model and the benchmark through, it can be represented as equation 2.9.

$$L\big(xz\big) = ||x - z||^2 \tag{2.9}$$

or Equation 2.10

$$L_H\big(x, z\big) = -\sum_{k}^{d} [x_k log z_k - \big(1 - x_k\big) log(1 - log z_k)] \tag{2.10}$$

The Equation 2.9 is the squared error objective for real value x and the Equation 2.10 is the cross-entropy objective for binary x[21]. Both methods attempt to minimize the loss function involve resampling the shuffled inputs and re-reconstructing the data, until it finds those inputs which bring its model closest to what it has been told is true. To illustrate the difference between the corrupted model and incorrupted model, we use Figure 2.6 to represent the model.

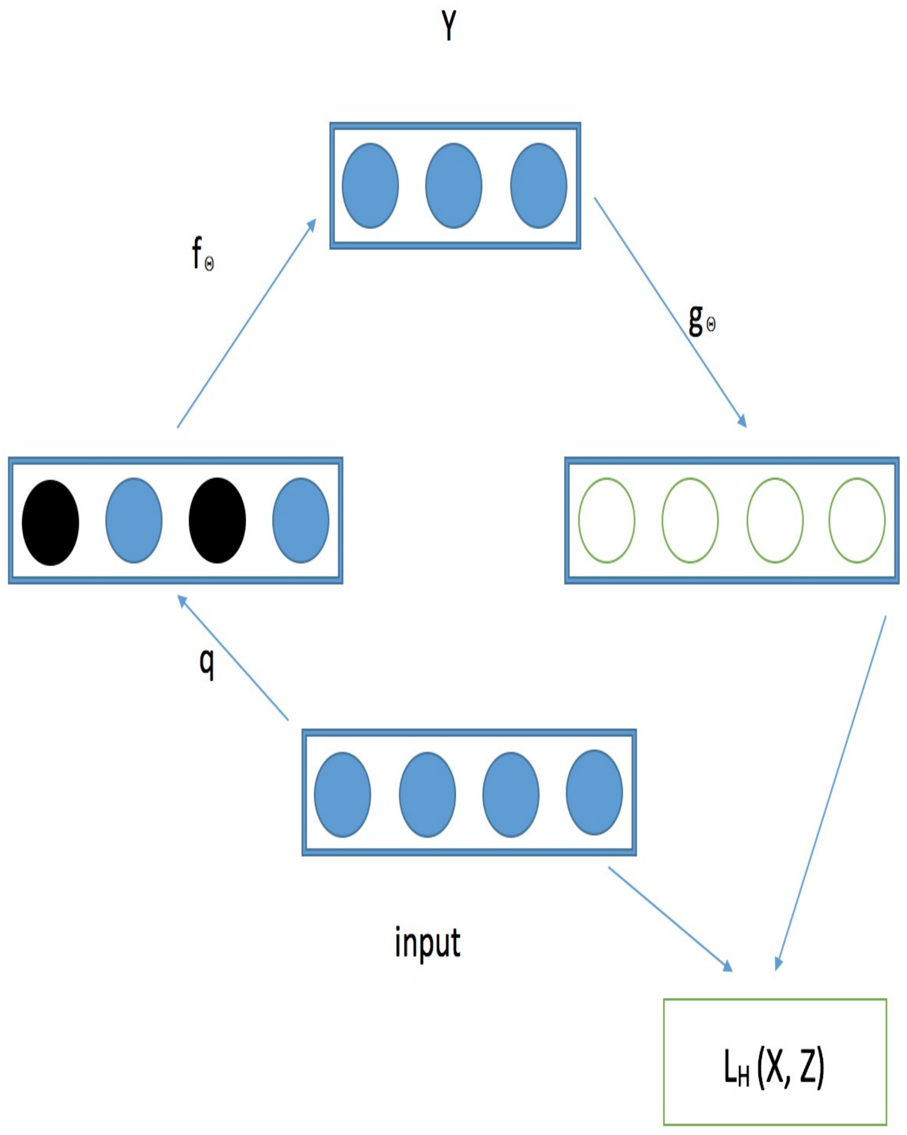


Figure 2.6: Example of an auto-encoder corruption model
Input nodes are corrupted via process q, then the encoder tries to map the corrupted input to Y via process f and Y can reconstruct via process g_θ. Then the reconstruction error is measured by $L_\theta(X, Z)$.

As shown in Figure 2.6, the raw input was corrupted via process q. The black node denotes the corrupted input. Then the corrupted input change to Y via process

f_θ. After that, the Y tries to reconstruct the raw input via process g_θ, Then we need to backpropagate through reconstruction error $L_H(X, Z)$.

2.7 Model for Gene Expression Prediction

By initialing random parameters, traditional MLP is not able to perform well by directly optimizing the supervised objective of interest (for example the log probability of correct classification) such as gradient descent.

A more efficient way to achieve a better performance is to use a a local unsupervised criterion to pre-train each layer in turn, and produce a useful higher-level representation from the lower-level representation output by the previous layer. Thus, the gradient descent on the supervised objective leads to much better solutions in terms of generalization performance[21].

The model we proposed for solving Gene Expression prediction problem is represented as Figure 2.7.

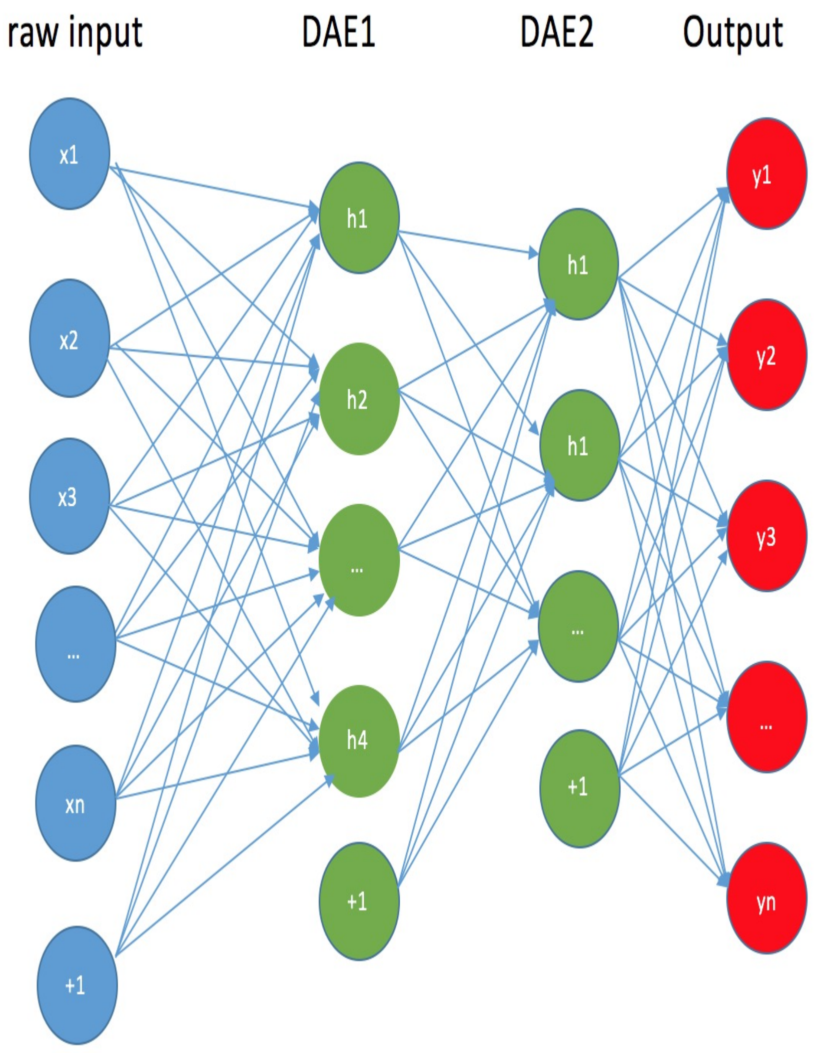


Figure 2.7: Model for MLP and DAE

The work flow chart for deep learning model is represented as Figure 2.8.

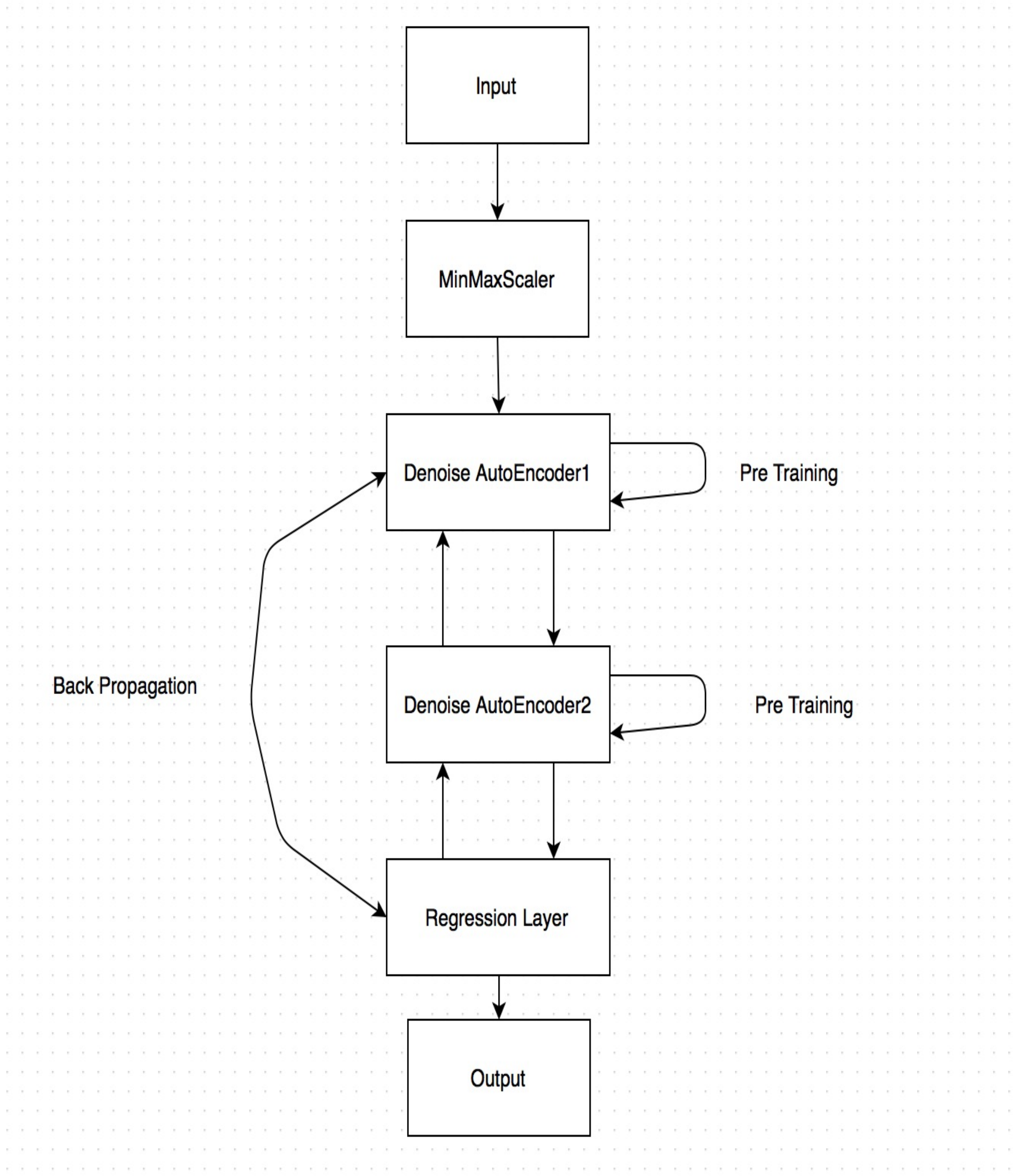


Figure 2.8: Completed work flow chart for our regression model

As the Figure 2.8 indicates, a raw input will be first processed by the MinMaxScaler. After that, the processed input is trained in first Auto-Encoder with back propagation, the output of the first auto-encoder is taken as input of the second Auto-Encoder and train with backpropagation as well. Then we will backpropagate the entire network as Multilayer perceptron.

By building a model with multilayer perceptron and stacked denoising autoencoder, we are apporaching the gold of extracting a higher level of useful features of raw input, and thus getting a accurate result.

2.8 Cross Validation

Cross-validation[22] is a model validation technique, sometimes called rotation estimation. Cross-validation is mainly used for assessing how the results of a statistical analysis will generalize to an independent data set. When the goal is prediction and someone wants to estimate how accurately a predictive model will perform in practice, cross validation is very useful. In a prediction problem, a model is usually given a data set of known data on which training is run (training dataset), and a dataset of unknown data (or first seen data) against which the model is tested (testing dataset).

The objectives of cross validation is to define a dataset so as to "test" the model in the training phase (i.e., the validation dataset), in order to limit problems like overfitting and to give an insight on how the model will generalize to an independent dataset (i.e., an unknown dataset, for instance from a real problem), etc.

In our approach, we split the dataset into three datasets, one is a training dataset and validation dataset to be used in training phase, the other one is a test dataset which does not participate in any training to avoid overfitting. The process is presented as Figure 2.9 below:

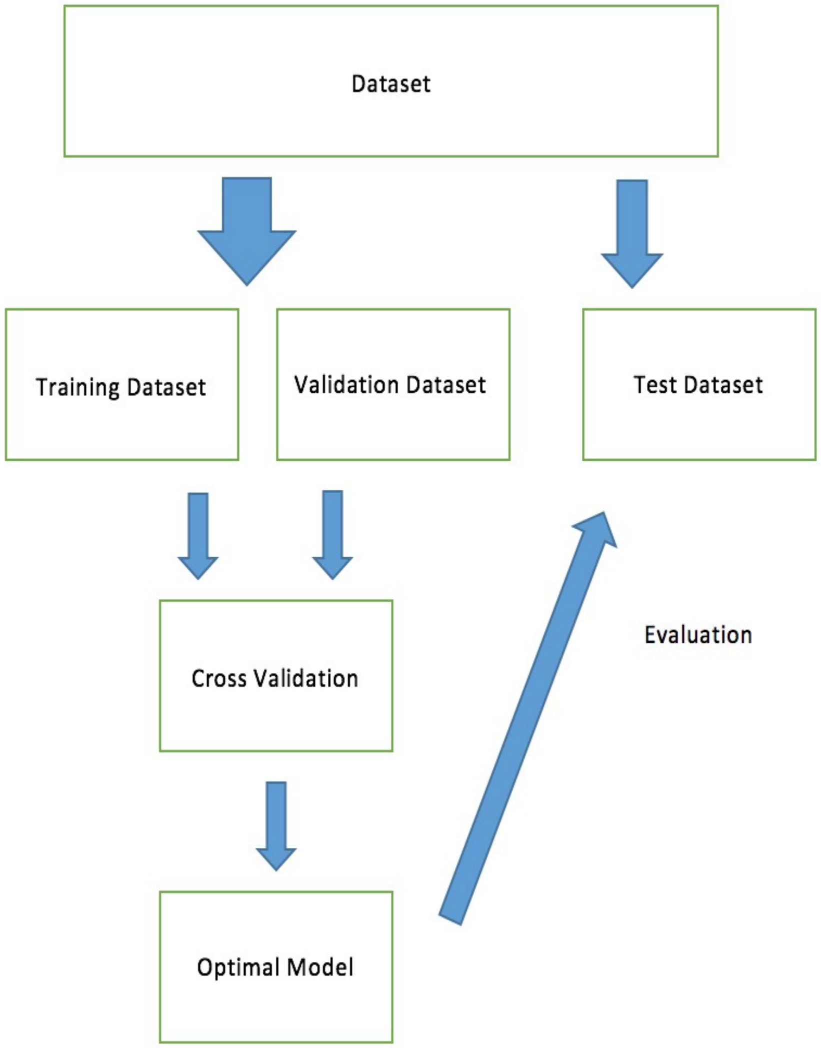


Figure 2.9: The process of performing cross validation

In our experiement, we will split the dataset into training and test data sets. Additionally, we will take part of the training dataset as a validation data set, which does not participate in training, and then used five folds of cross validation on the training dataset to obtain our optimal model.

Finally, we will apply the model to an independent test dataset to evaluate the performance. In addition, we will take multiple experiments to obtain results in order to avoid randomness of prediction.

2.9 Result

This section compares the results of Lasso and Random Forest implementation.

We will use mean square error (MSE) to evaluate the performance of our model. The Equation 2.11 shows how to calculate MSE.

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i' - y_i)^2 \tag{2.11}$$

To avoid overfitting, we separated the dataset into two folds. One data set fold was used for training, and the other was used as a test dataset, which was not used for training. The result of our model were obtained for a test dataset to compare the results obtained on different methods.

2.9.1 Results for Applying Yeast Data Set

The first method we used was Lasso. After cross validation, we used the optimal model trained to make predictions on our test dataset.

The follwing Table 2.1 shows the results obtained for Lasso.

Alpha	MSE
0.05	0.3516
0.1	0.3182
0.2	0.3002
0.3	0.2951
0.4	0.2930
0.5	0.2918
0.6	0.2914
0.7	0.2912
0.8	0.2912

Table 2.1: Results obtained for Lasso

We also used a Figure 2.10 to show MSE obtained with different alpha for LASSO.

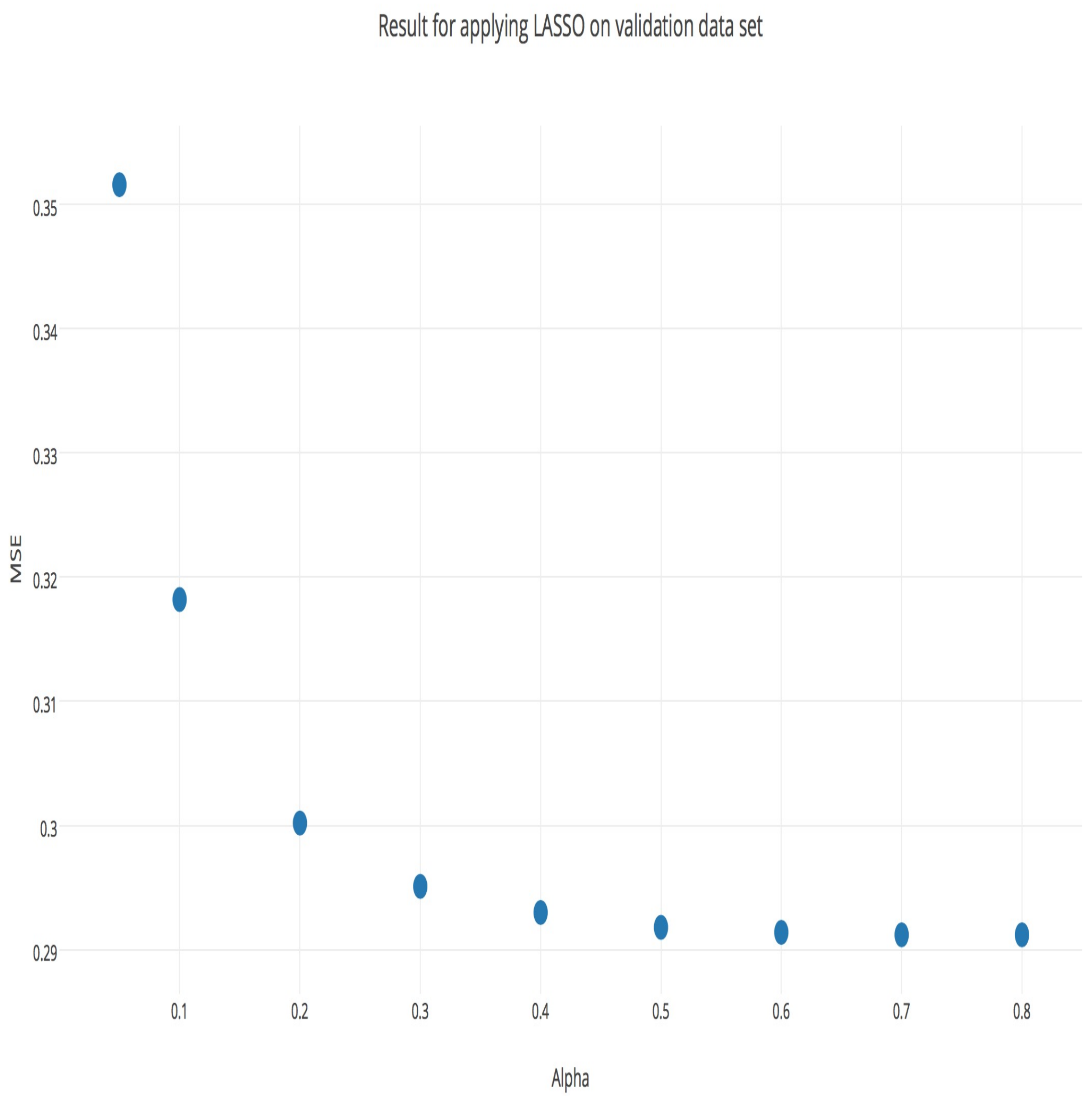


Figure 2.10: Results after applying MLP with SAE on validation data set
α is a constant that multiplies the L1 term.

The second method we adopted was Random Forests. After cross validation, the follwing Table 2.2 shows the results obtained from Random Forests after cross validation.

Estimators	MSE
10	0.3221
20	0.3127
30	0.3080
40	0.3001
50	0.2989
60	0.3003
70	0.2986
100	0.3003
150	0.2974
200	0.2967

Table 2.2: Results obtained for Random Forests

The following Figure 2.11 shows the results obtained from Random Forests implementation.

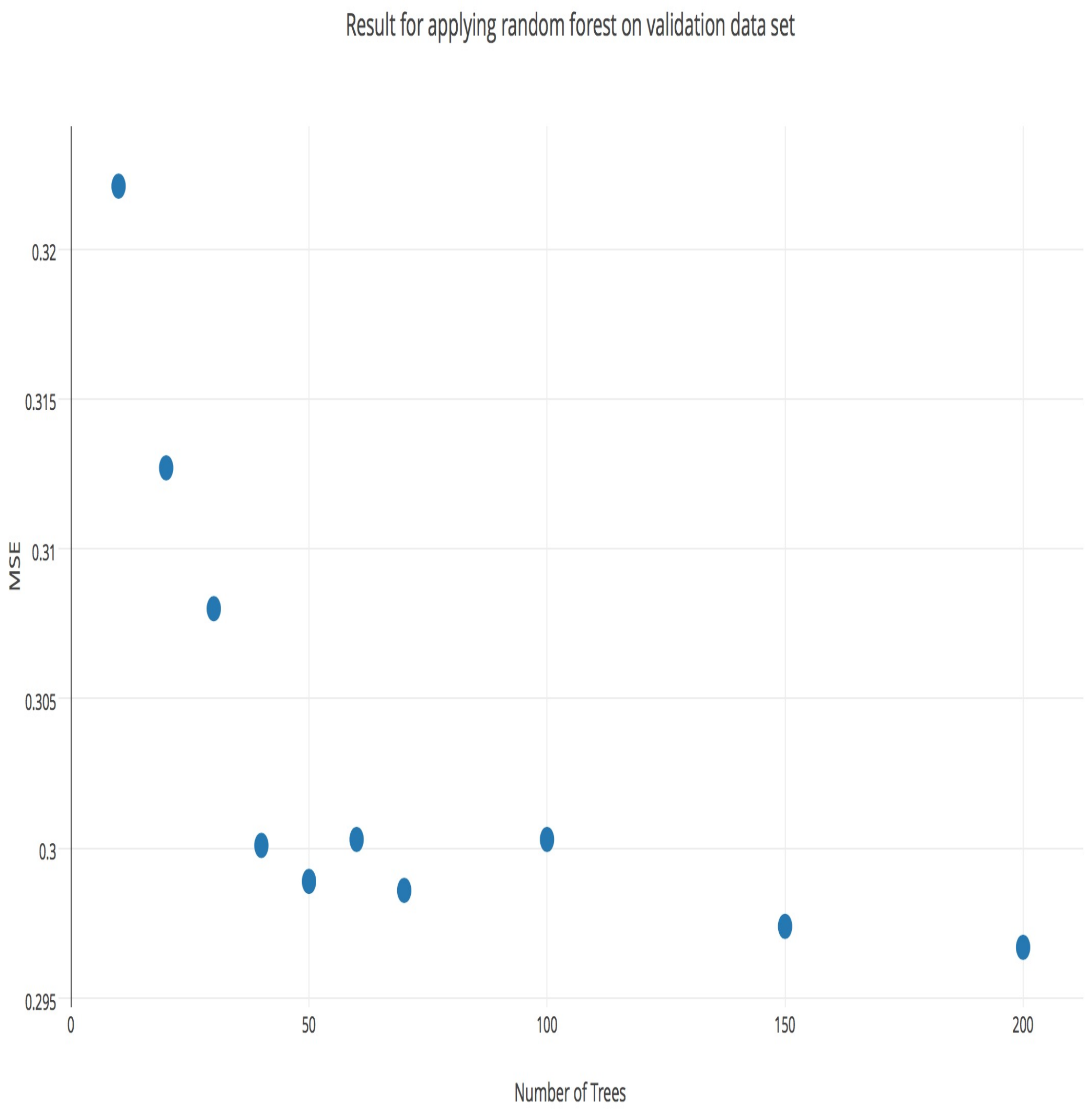


Figure 2.11: Results after applying Random Forests on validation data set
The number of estimators denotes the number of trees in the forest.

Finally, we used our model (Multilayer Perceptron with Stacked Denoising Auto-encoder) on the same test dataset. The Table 2.3 shows detailed results,

Learning Rate	MSE
0.1	0.289001373
0.01	0.290938859
0.001	0.289527237
0.0001	0.290783006
0.00001	0.29175354

Table 2.3: Cross validation result obtained using MLP with SAE

The Figure 2.12 illustrates the results obtained:

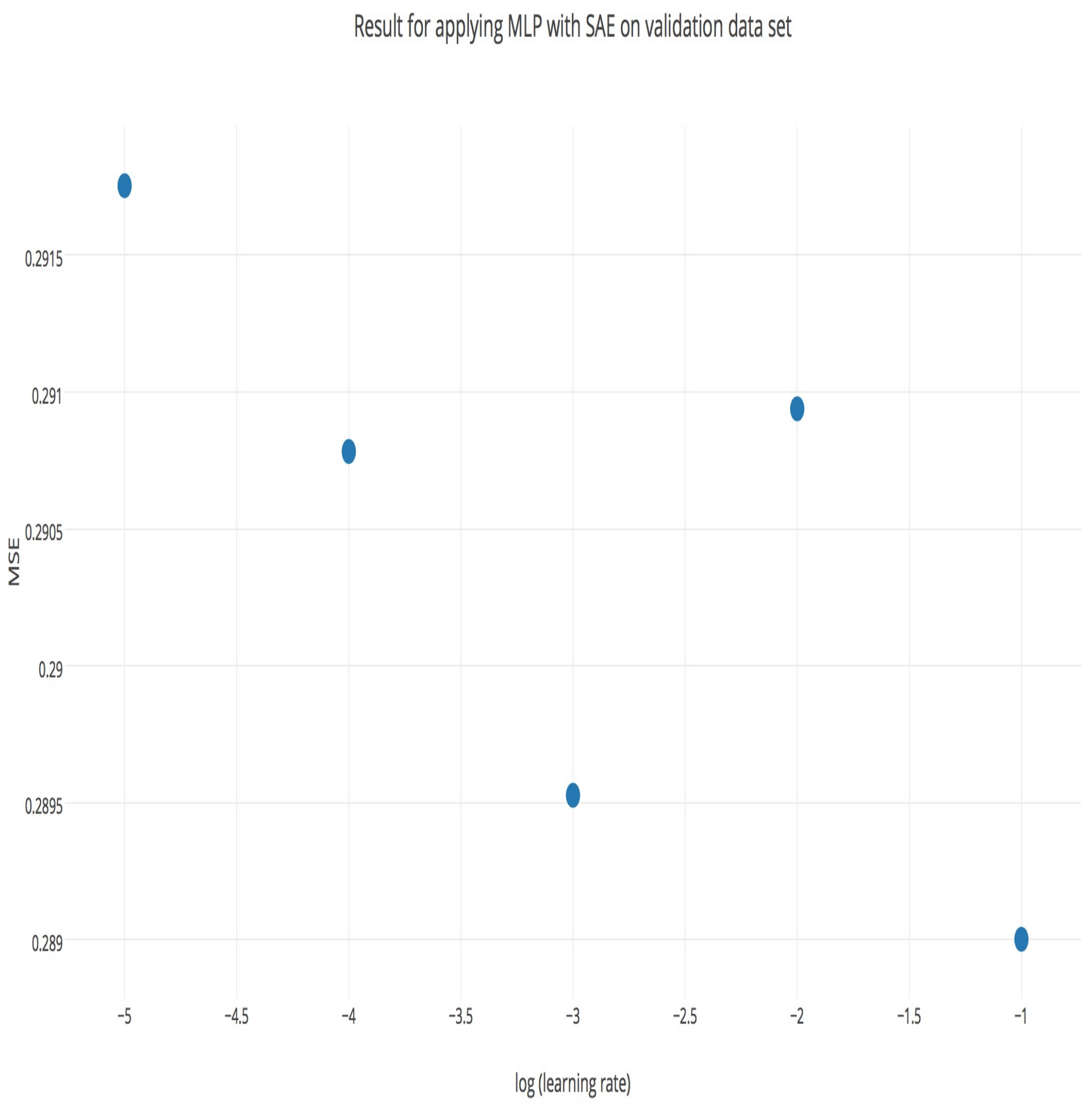


Figure 2.12: Results after applying MLP with SAE on validation data set

Finally, we use used the above plot and Table 2.4 to show the best results for the three method, which shows that the deep learning method produce comparable results when compared with other methods on the test data set.

Methods	MSE
MLP with SAE	0.3093
Lasso	0.3030
Random Forests	0.3107

Table 2.4: Result obtained for Three Methods

Figure 2.13 is used to illustrate the results of different methods.

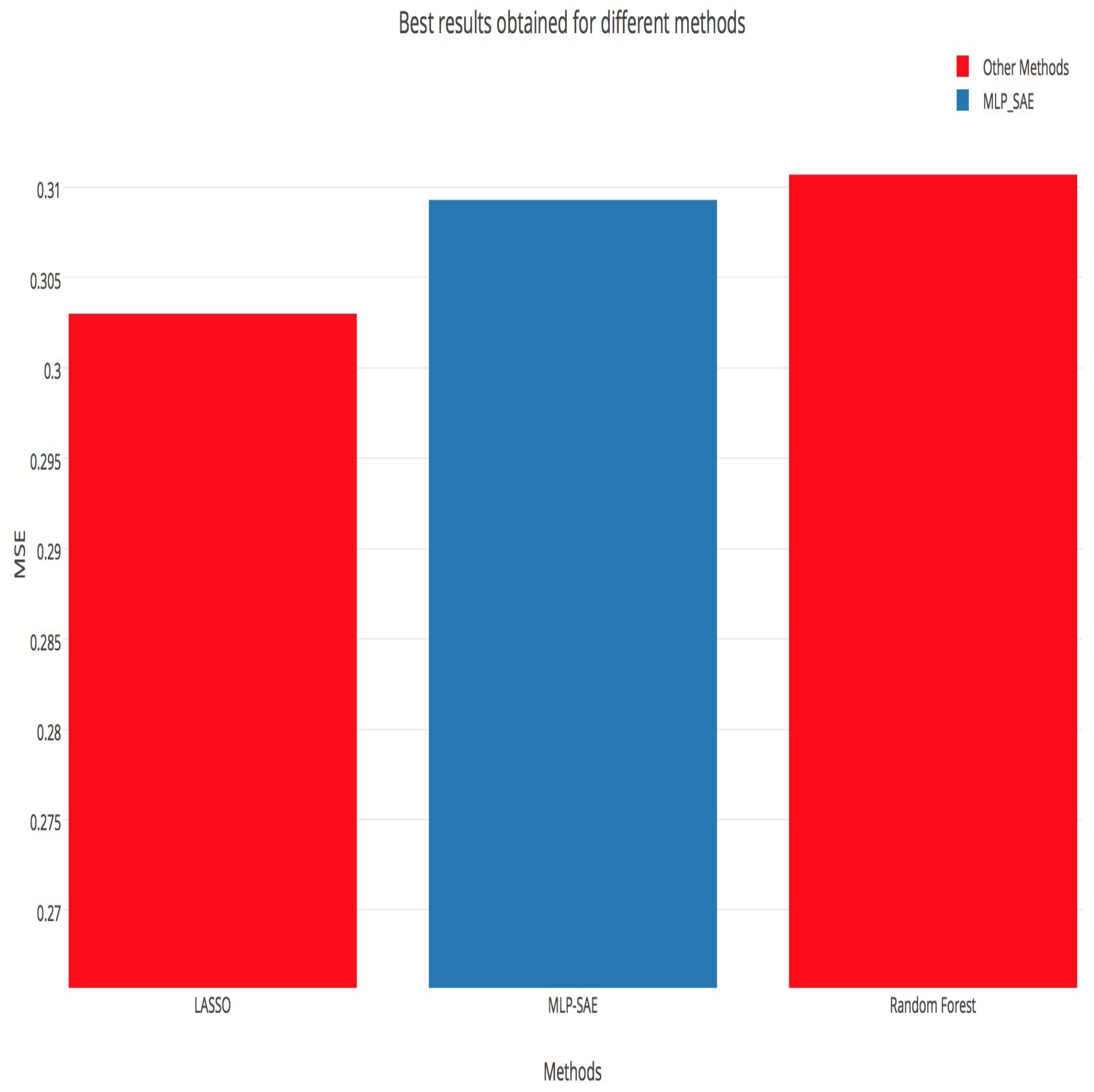


Figure 2.13: Comparison between Methods

Next, we use Figure 2.14 to find out how well the model predicted the gene expression.

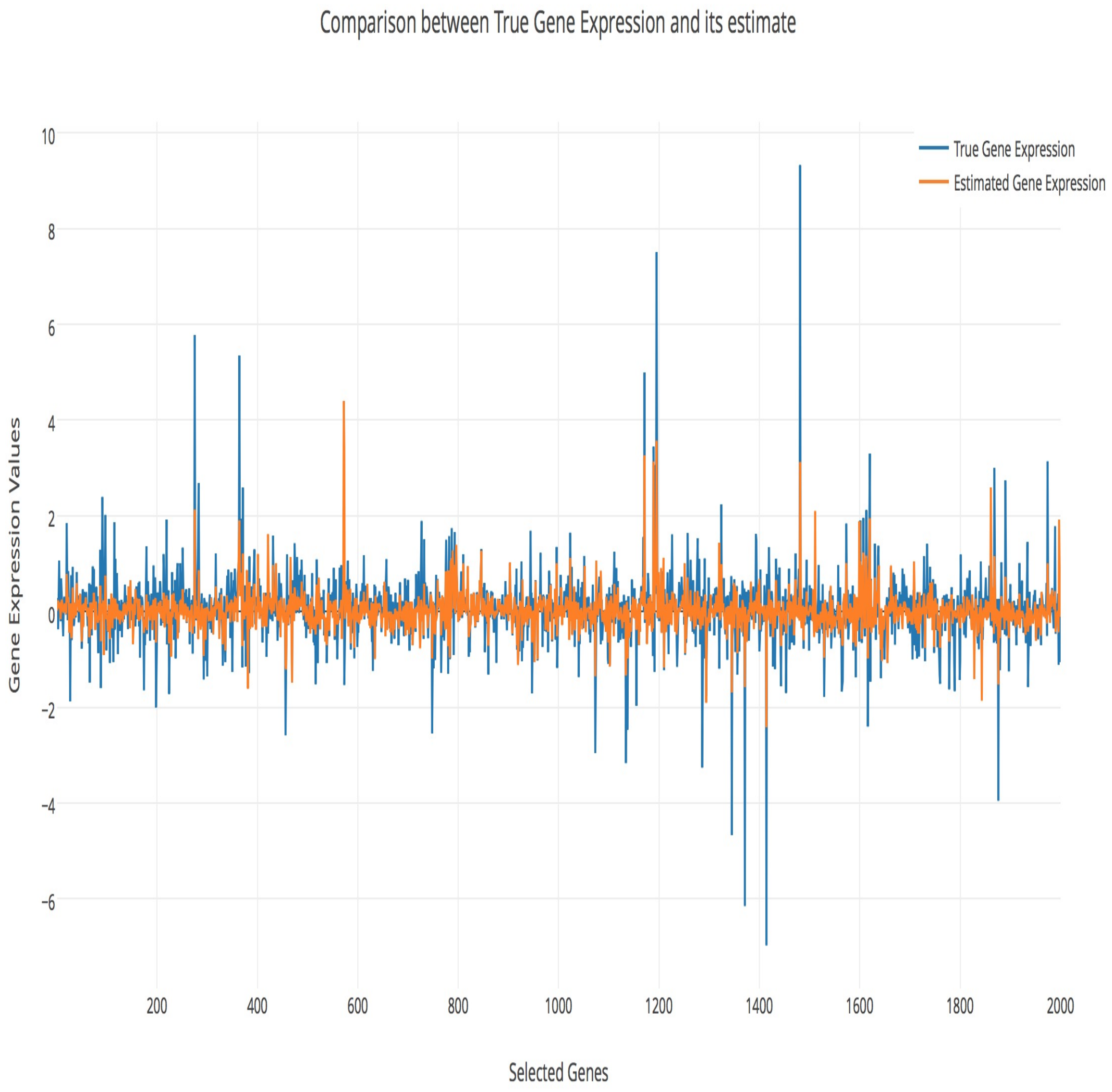


Figure 2.14: Comparison between True Gene Expression and Estimated Expression

Then we to show the best results which follows as Figure 2.15.

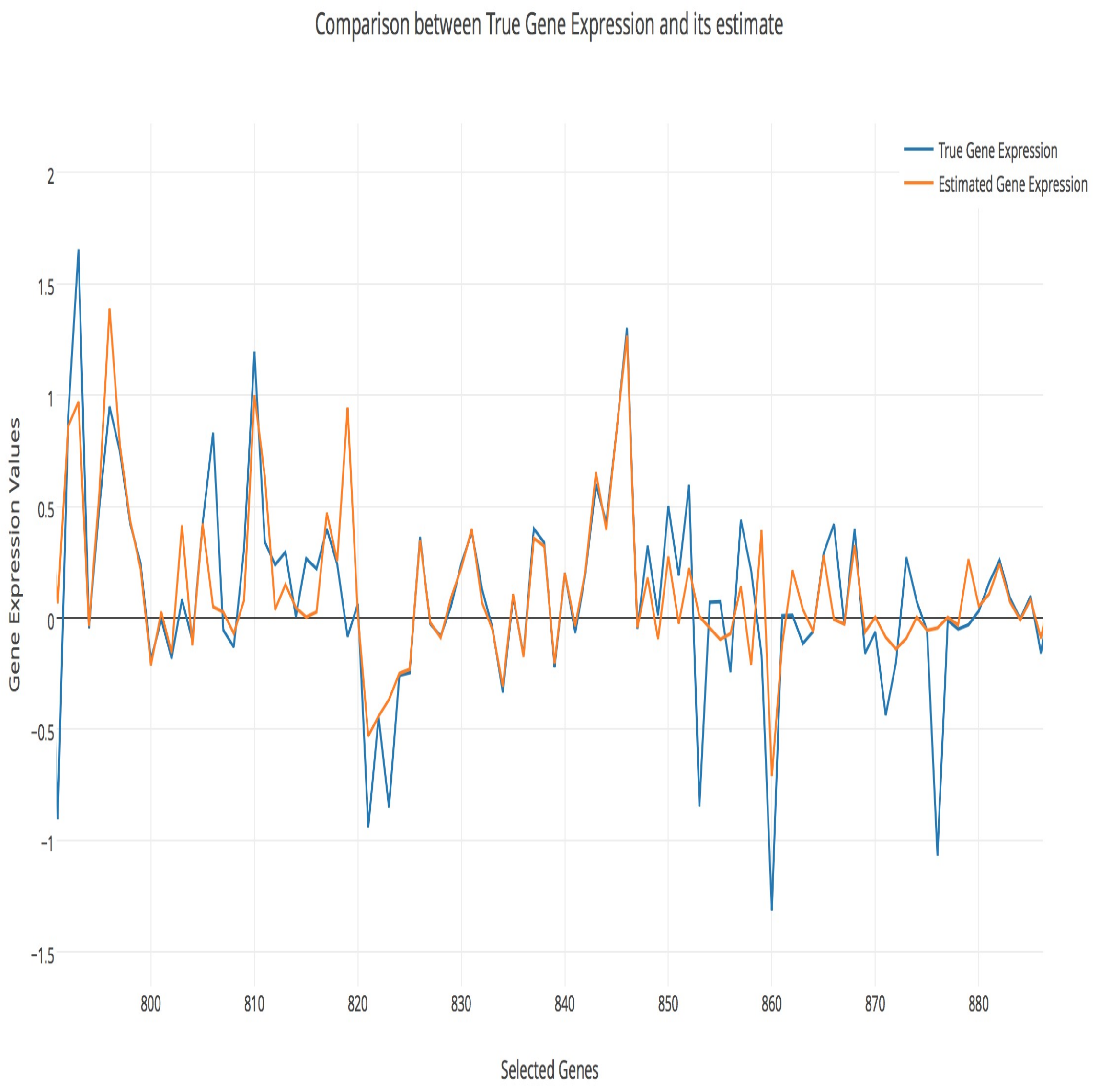


Figure 2.15: Comparison between True Gene Expression and Estimated Expression

Figure2.16 shows the average MSE is about 0.3 for most of the prediction, which shows that our prediction are close the to the MSE except a few genes.

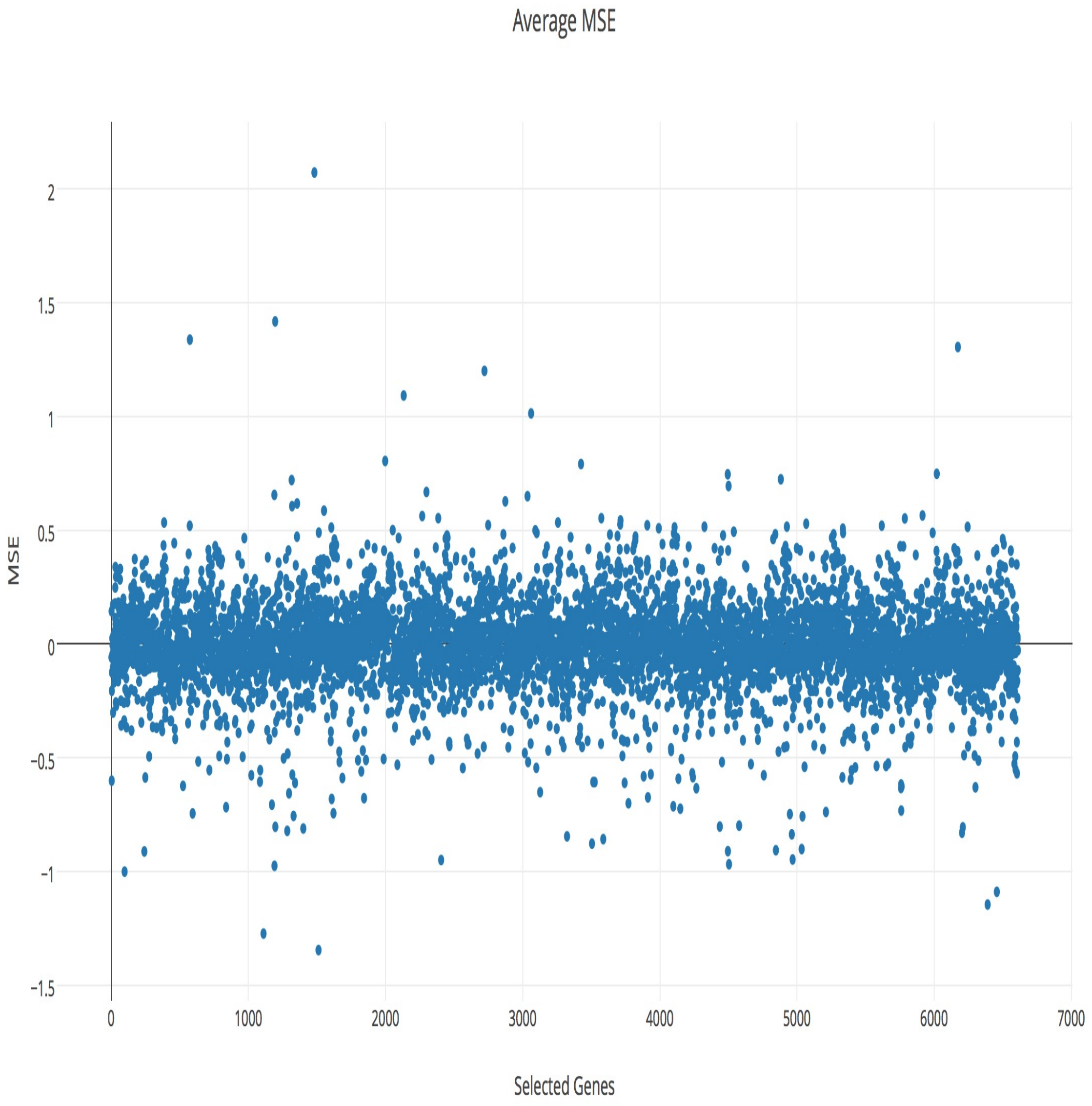


Figure 2.16: Average MSE for predictions

2.9.2 Conclusions

Development of a method capable of predicting accurate results has a lot of difficuli-ties.

In this chapter, we presented a new solution based on Multilayer Perceptron and Stacked Denoise Auto-Encoder that generated results very close to the results of existing approaches including Lasso and Random Forests when testing independent data sets. Even though the results are encouraging, there is room for improvement of the model. In next chapter, we will discuss the improvement of our model by using Dropout.

Chapter 3

Improving Model by Applying Dropout

3.1 Abstract

Although multilayer perceptron combined with a stacked denoising autoencoder can produce comparable results when compared withother methods, we still needed to better control overfitting in order to improve the performance. We tested Dropout[23] to see if it could handle overfitting and thus obtain better results.

We found that the previous deep learning model with Dropout yields a better result compared to previous results without Dropout.

3.2 Introduction to Dropout

Dropout prevents over fitting and provides a way of approximately combining exponentially many different neural network architectures efficiently [23].

The term dropout refers to dropping out units (hidden and visible) in a neural network. To drop a unit out is to temporarily remove it from the network, along with all its incoming and outgoing connections. The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.

With the above in mind, the intuitive goal of drop-out regularization is to approximate the following concept: Ignoring units and their associated weights by a probability p for a particular training sample, train with back propagation. Then, repeat (ignoring any other random set of units, then train) and train training samples. Average the weights across all these modified structures when doing predictions on new samples. Figure 3.1 shows an example of a neural network with Dropout:

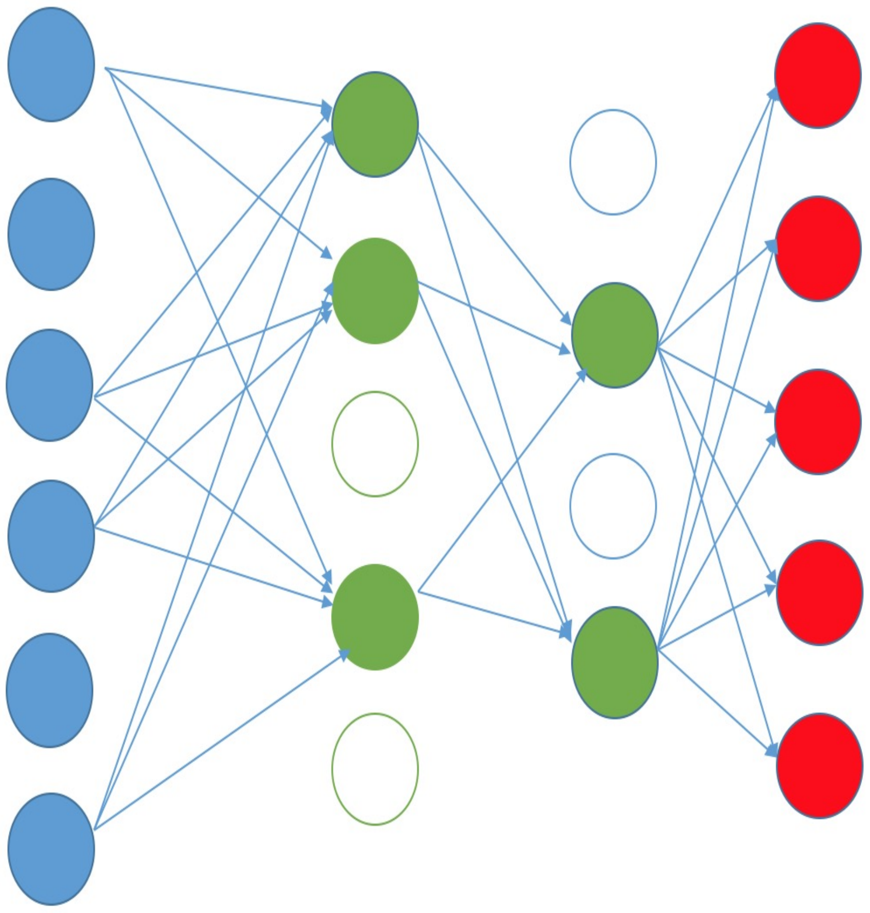


Figure 3.1: An example of a neural network with Dropout

3.3 Results and Discussion

We used our model as developed with Dropout (Multilayer Perceptron with Stacked Denoise Auto-encoder and Dropout) on the independent test dataset following the same protocol as in previous experiments.

3.3.1 Results for Applying Yeast Data Set

This section showst he same training data as previously used being applied to our model with Dropout. Table 3.1 shows the result obtained for different methods.

Methods	MSE
MLP with SAE and Dropout	0.3082
MLP with SAE	0.3093
Lasso	0.3030
Random Forests	0.3107

Table 3.1: Results obtained using different methods

The Figure 3.2 illustrates results in different learning rates.

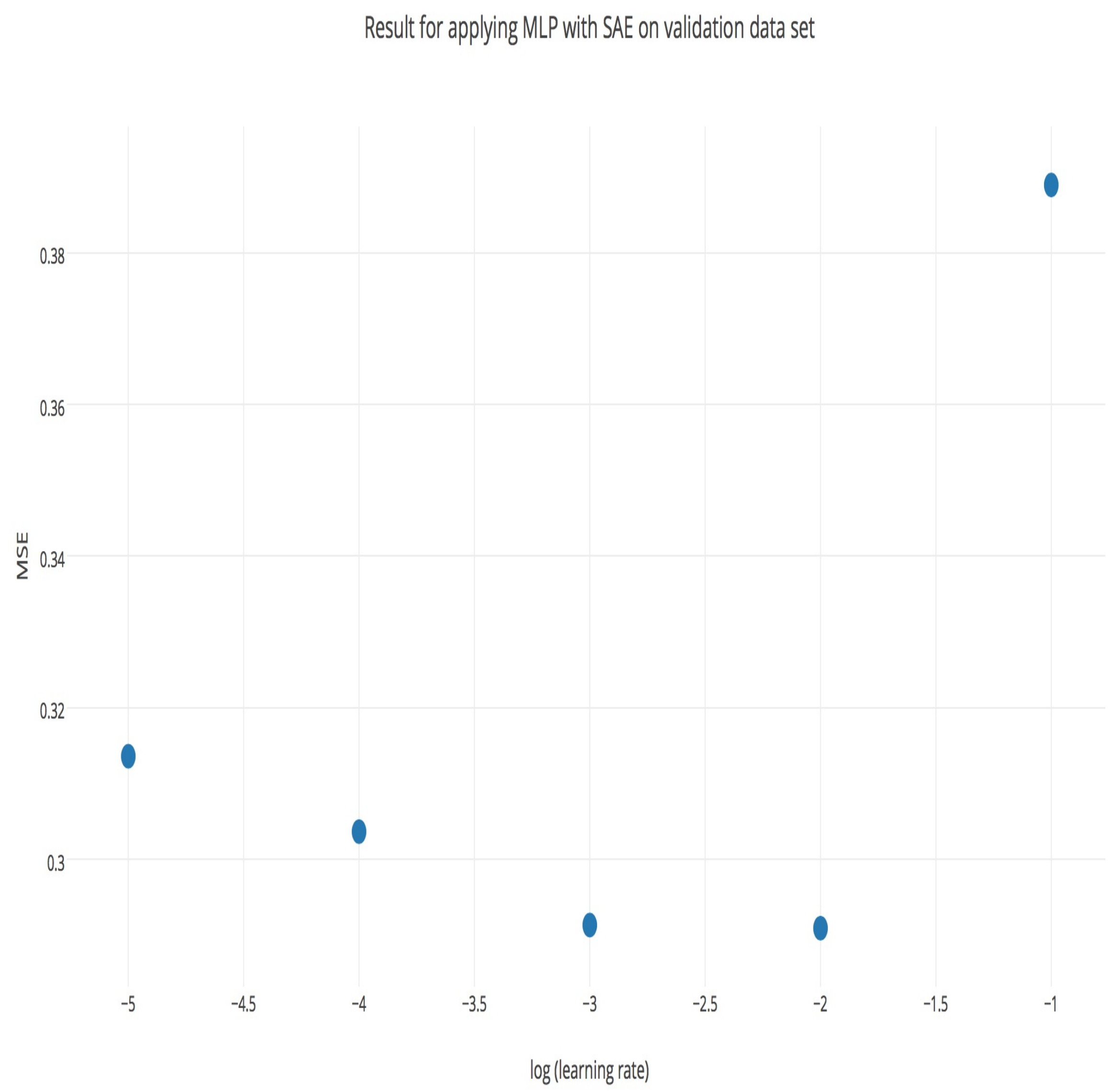


Figure 3.2: Result for applying MLP with SAE and Dropout on validation dataset

Then we compared results obtained with results of last chapter, showing that the model with Dropout outperformed its previous models on test data sets as shown by Figure 3.3.

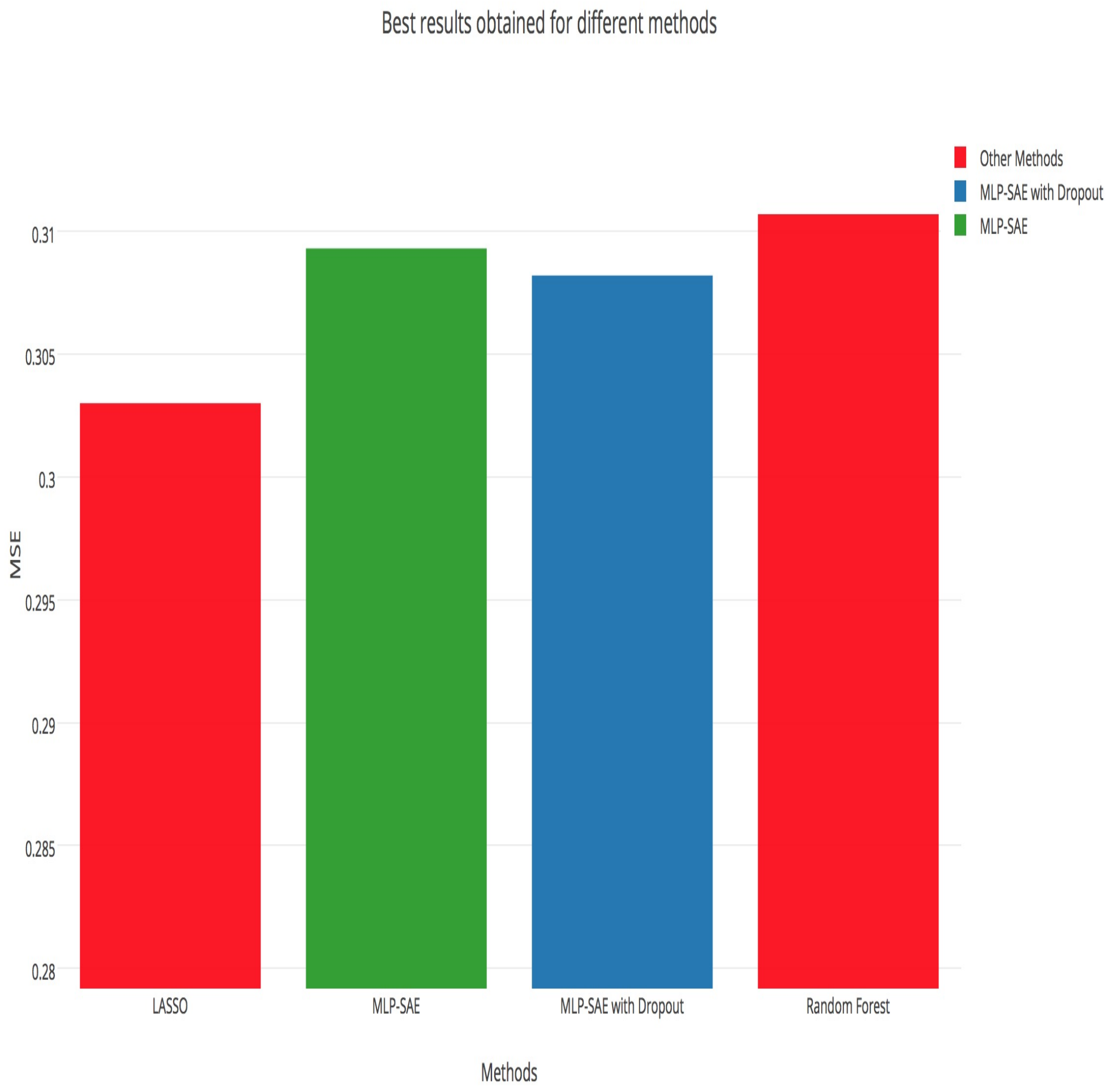


Figure 3.3: Comparison between methods

To further interpret the results, we can use Figure 3.4 to show that the deep learning model is capable of accurately predicting gene expressions.

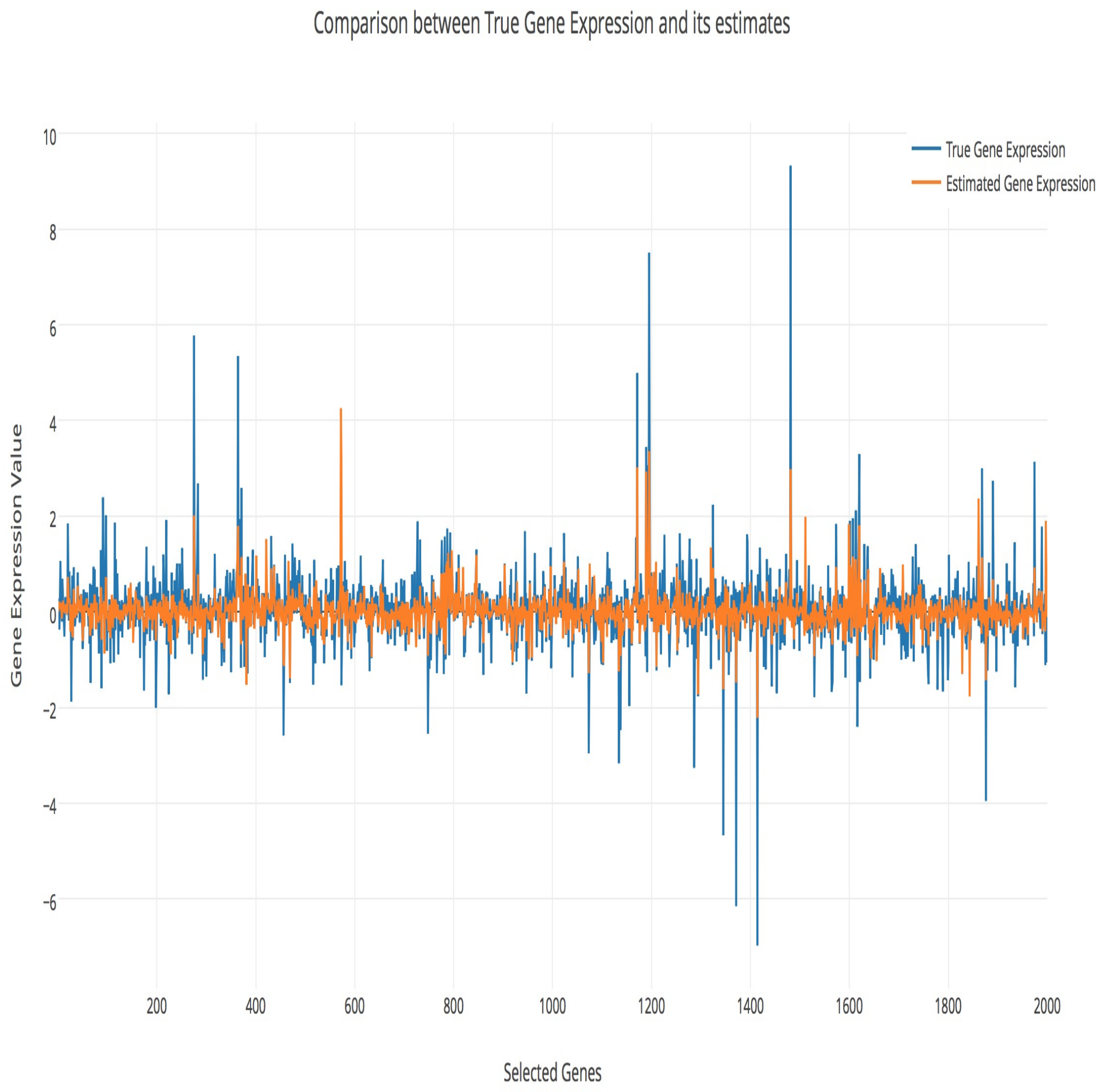


Figure 3.4: Comparison between estimated expression and true gene expression for 2000 genes

Figure 3.5 shows part of genes that can be well predicted.

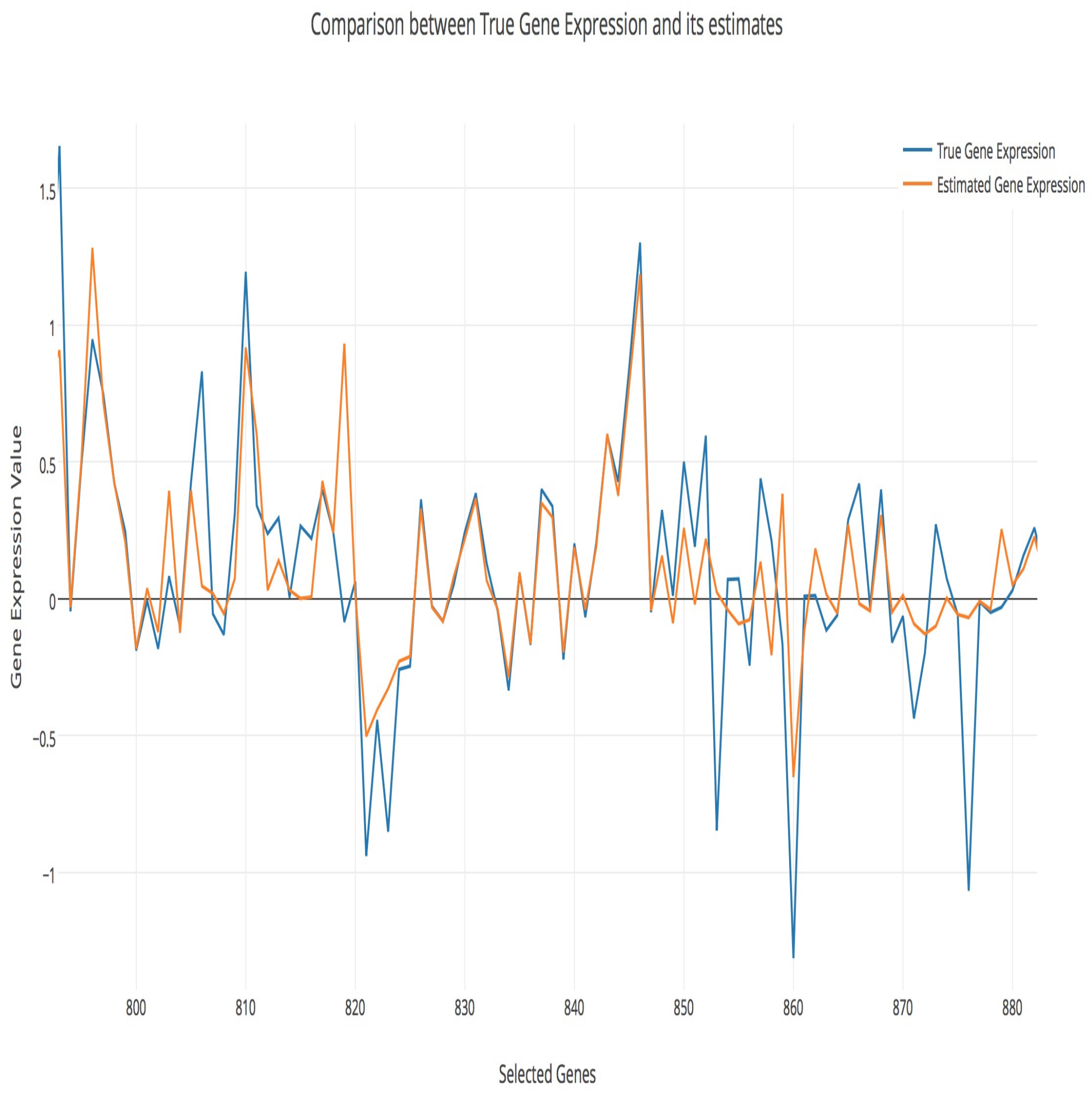


Figure 3.5: estimated expression and true gene expression for 100 genes

3.3.2 Conclusion

This chapter shows how we improved our deep learning model by adding a Dropout technique, which proved to be an effective method to prevent overfitting and enhance performance.

In our experiments, results of a new model with Dropout is better than the result of previous deep learning model, which yielded an outcome closer to the outcome of other popular methods. The results showed that using Dropout is a feasible way to handle overfitting in gene expression prediction.

Chapter 4

Summary and Concluding Remarks

This chapter summarizes the techniques used in training the deep learning model as well as Dropout to improve the model. Limitations and future work are also discussed.

4.1 Summary

Predicting gene expression levels is important in biological systems. Due to missing data and difficulty in extracting useful features and overfitting, a new prediction model based on deep learning has been developed.

We used Sciki-Learn toolkit[24] such as MinMAXScaler and Imputer to handle missing data from genotype features in order to minimize their negative effect during training.

Next, we used stacked denoising auto-encoder to train our regression model in order to extract useful features, then used multilayer perceptron for backpropagation and obtained a result close to results of the other methods.

In conclusion, gene expression can be affected by a group of gene variations. The Lasso method can be the most suitable method since it can shrink some covariance to exactly zero. Deep learning cannot outperform because it contributes to the global effect of all variances and sometime it overconsiders the effect and amount of noise added. Thus, Dropout can well control overfitting thereby improving the results. Random Forests behave since it randomly chooses some samples and if the sample selection is bad, then its performance is poor.

All the successful experiments indicated that Deep Learning has great potential in solving problems in biological systems.

4.2 Limitations and Future Work

Although the results has shown that the Deep Learning models we developed are as good as the other methods, we may explore more possibilities for those untested data since there are many type of Genotype data and the yeast data set is just one of them.

In addition, there are many deep learning architectures such as Restricted Boltzmann Machine [25] and the Recurrent Neural Network [26], which can be used for solving gene expression prediction problems. Thus there is huge room for improvement when using deep learning as a solution.

Meanwhile, other data processing methods must be explored to find solutions that are more suitable for handling missing values.

Appendix A

Supplementary Information

The package we used to pre-process data is scikit-learn[24], which is a Python library for processing data.

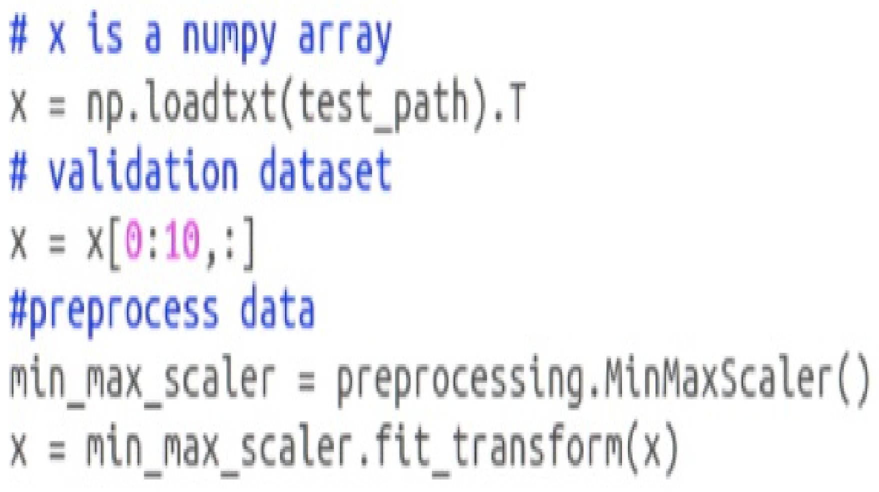
The prediction model we have made is based on the Pylearn2[27] Package. Here we will give an overview of Pylearn2 and how we used it to customize our model.

A.1 Introduction to scikit-learn

Scikit-learn (formerly scikits.learn) is an open source machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

A.1.1 MinMaxScaler

To process the raw data, we use the toolkit provided by Scikit-learn, which is called MinMaxScaler[13]. The Figure A.1 shows the code of implementing Data Pre-process using MinMaxScaler.



```
# x is a numpy array
x = np.loadtxt(test_path).T
# validation dataset
x = x[0:10,:]
#preprocess data
min_max_scaler = preprocessing.MinMaxScaler()
x = min_max_scaler.fit_transform(x)
```

Figure A.1: Code of MinMaxScaler

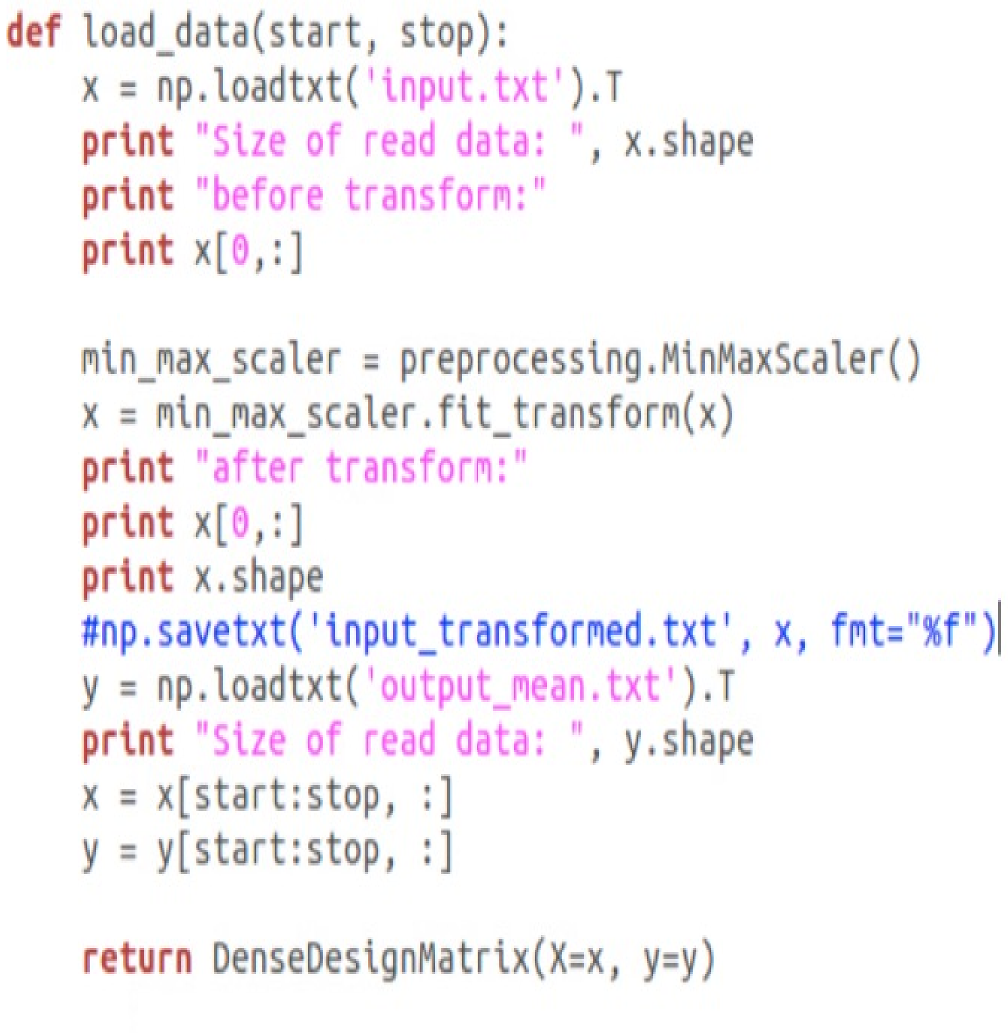
A.2 Introduction to Pylearn2

Pylearn2 is an open source software library. It is built on top of Theano[28] [29] and is designed for researchers studying machine learning to be able to easily configure advanced machine learning experiments. The library divides most machine learning problems into three parts: the data set, the model, and the training algorithm. The training algorithm works to adapt the model to fit the values provided in the data set.

A.2.1 Wrapping up Data

We need to write a small Python wrapper to put the data in the right format for Pylearn2. We can do this by creating an instance of Pylearn2s DenseDesignMatrix class, which is used to store simple datasets, where the dataset of features can be represented as a single matrix with examples in rows and features in columns.

This class expects the features (genotype) in a matrix X and the targets (gene expression value) in a matrix Y. Figure A.2 the implementation of the wrapper:

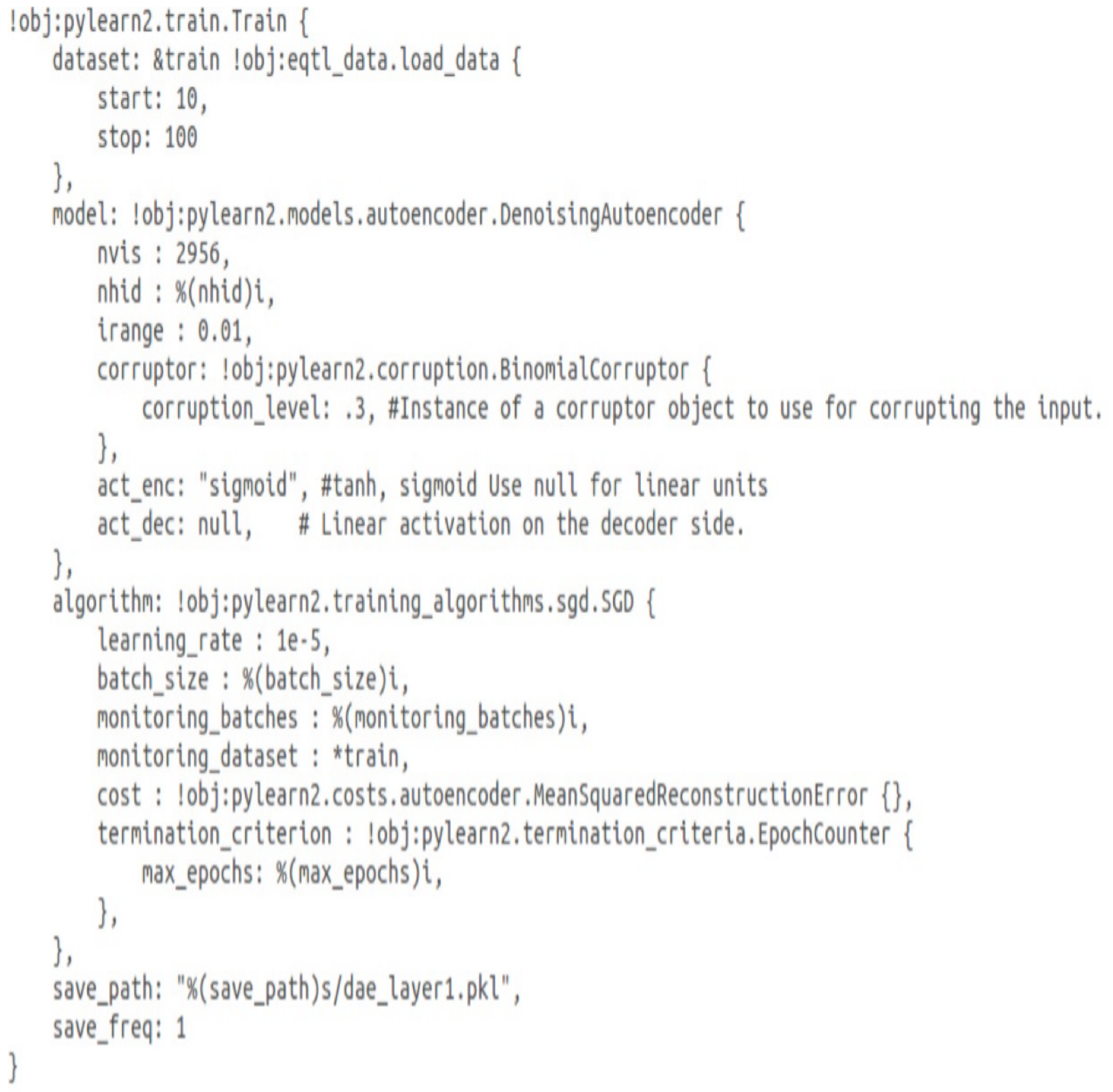


```
def load_data(start, stop):
    x = np.loadtxt('input.txt').T
    print "Size of read data: ", x.shape
    print "before transform:"
    print x[0,:]

    min_max_scaler = preprocessing.MinMaxScaler()
    x = min_max_scaler.fit_transform(x)
    print "after transform:"
    print x[0,:]
    print x.shape
    #np.savetxt('input_transformed.txt', x, fmt="%f")
    y = np.loadtxt('output_mean.txt').T
    print "Size of read data: ", y.shape
    x = x[start:stop, :]
    y = y[start:stop, :]

    return DenseDesignMatrix(X=x, y=y)
```

Figure A.2: Implementation of Data Wrapper

A.2.2 Define YAML File

We now need to write a YAML file describing the regression problem to Pylearn2. Most of the process is already described in the existing tutorials for Pylearn2. We will focus on how to solve a regression problem. One way to do this is with a deep neural net, or multilayer perceptron (MLP). Figure A.3 shows an YAML file showing how to define the first autoencoder for our model:



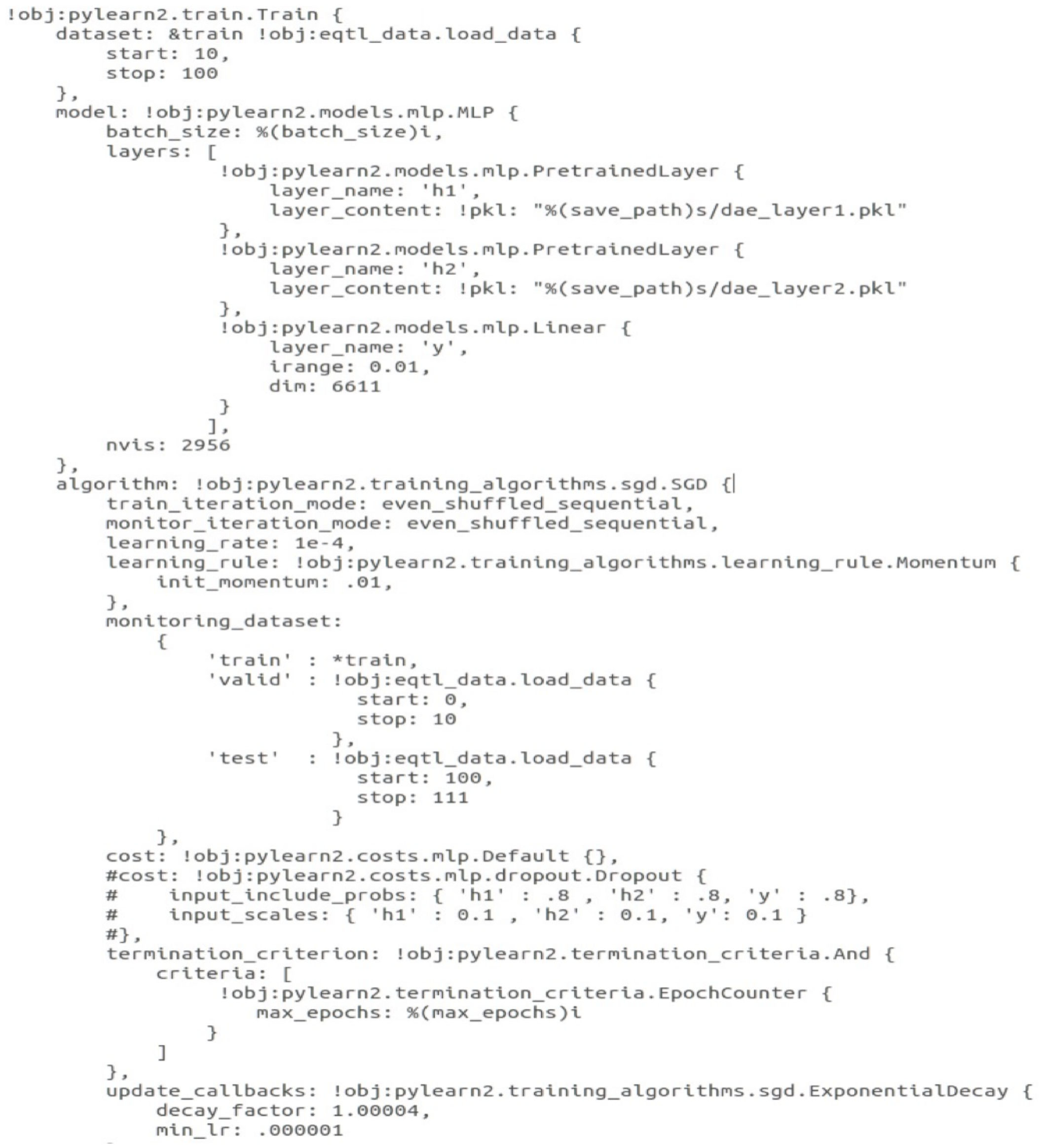
```
!obj:pylearn2.train.Train {
    dataset: &train !obj:eqtl_data.load_data {
        start: 10,
        stop: 100
    },
    model: !obj:pylearn2.models.autoencoder.DenoisingAutoencoder {
        nvis : 2956,
        nhid : %(nhid)i,
        irange : 0.01,
        corruptor: !obj:pylearn2.corruption.BinomialCorruptor {
            corruption_level: .3, #Instance of a corruptor object to use for corrupting the input.
        },
        act_enc: "sigmoid", #tanh, sigmoid Use null for linear units
        act_dec: null,    # Linear activation on the decoder side.
    },
    algorithm: !obj:pylearn2.training_algorithms.sgd.SGD {
        learning_rate : 1e-5,
        batch_size : %(batch_size)i,
        monitoring_batches : %(monitoring_batches)i,
        monitoring_dataset : *train,
        cost : !obj:pylearn2.costs.autoencoder.MeanSquaredReconstructionError {},
        termination_criterion : !obj:pylearn2.termination_criteria.EpochCounter {
            max_epochs: %(max_epochs)i,
        },
    },
    save_path: "%(save_path)s/dae_layer1.pkl",
    save_freq: 1
}
```

Figure A.3: The YAML file generated for first layer of DNN model

A.2.3 Customizing Deep Learning Architecture

Some multilayer perceptrons output an estimate of a discrete variable, such as which category the input belongs to. For regression, we want to output an estimate of a continuous valued variable, which in this case is the gene expression value. To do this, we need the final layer of the multilayer perceptron to be one that models a continous layer.

Pylearn2 provides the MLP layer, which is called Linear. These layers can also be used as hidden layers so they are not singled out in a regression-specific module or anything like that. We also write our own layer using Stacked Denoise Auto-Encoders as pretrained layers. The following Figure A.4 shows how a YAML file configures an experiment of using MLP with Stacked Autoencode:



```
!obj:pylearn2.train.Train {
    dataset: &train !obj:eqtl_data.load_data {
        start: 10,
        stop: 100
    },
    model: !obj:pylearn2.models.mlp.MLP {
        batch_size: %(batch_size)i,
        layers: [
                 !obj:pylearn2.models.mlp.PretrainedLayer {
                     layer_name: 'h1',
                     layer_content: !pkl: "%(save_path)s/dae_layer1.pkl"
                 },
                 !obj:pylearn2.models.mlp.PretrainedLayer {
                     layer_name: 'h2',
                     layer_content: !pkl: "%(save_path)s/dae_layer2.pkl"
                 },
                 !obj:pylearn2.models.mlp.Linear {
                     layer_name: 'y',
                     irange: 0.01,
                     dim: 6611
                 }
                ],
        nvis: 2956
    },
    algorithm: !obj:pylearn2.training_algorithms.sgd.SGD {
        train_iteration_mode: even_shuffled_sequential,
        monitor_iteration_mode: even_shuffled_sequential,
        learning_rate: 1e-4,
        learning_rule: !obj:pylearn2.training_algorithms.learning_rule.Momentum {
            init_momentum: .01,
        },
        monitoring_dataset:
            {
                'train' : *train,
                'valid' : !obj:eqtl_data.load_data {
                            start: 0,
                            stop: 10
                           },
                'test'  : !obj:eqtl_data.load_data {
                            start: 100,
                            stop: 111
                           }
            },
        cost: !obj:pylearn2.costs.mlp.Default {},
        #cost: !obj:pylearn2.costs.mlp.dropout.Dropout {
        #    input_include_probs: { 'h1' : .8 , 'h2' : .8, 'y' : .8},
        #    input_scales: { 'h1' : 0.1 , 'h2' : 0.1, 'y': 0.1 }
        #},
        termination_criterion: !obj:pylearn2.termination_criteria.And {
            criteria: [
                 !obj:pylearn2.termination_criteria.EpochCounter {
                    max_epochs: %(max_epochs)i
                }
            ]
        },
        update_callbacks: !obj:pylearn2.training_algorithms.sgd.ExponentialDecay {
            decay_factor: 1.00004,
            min_lr: .000001
```

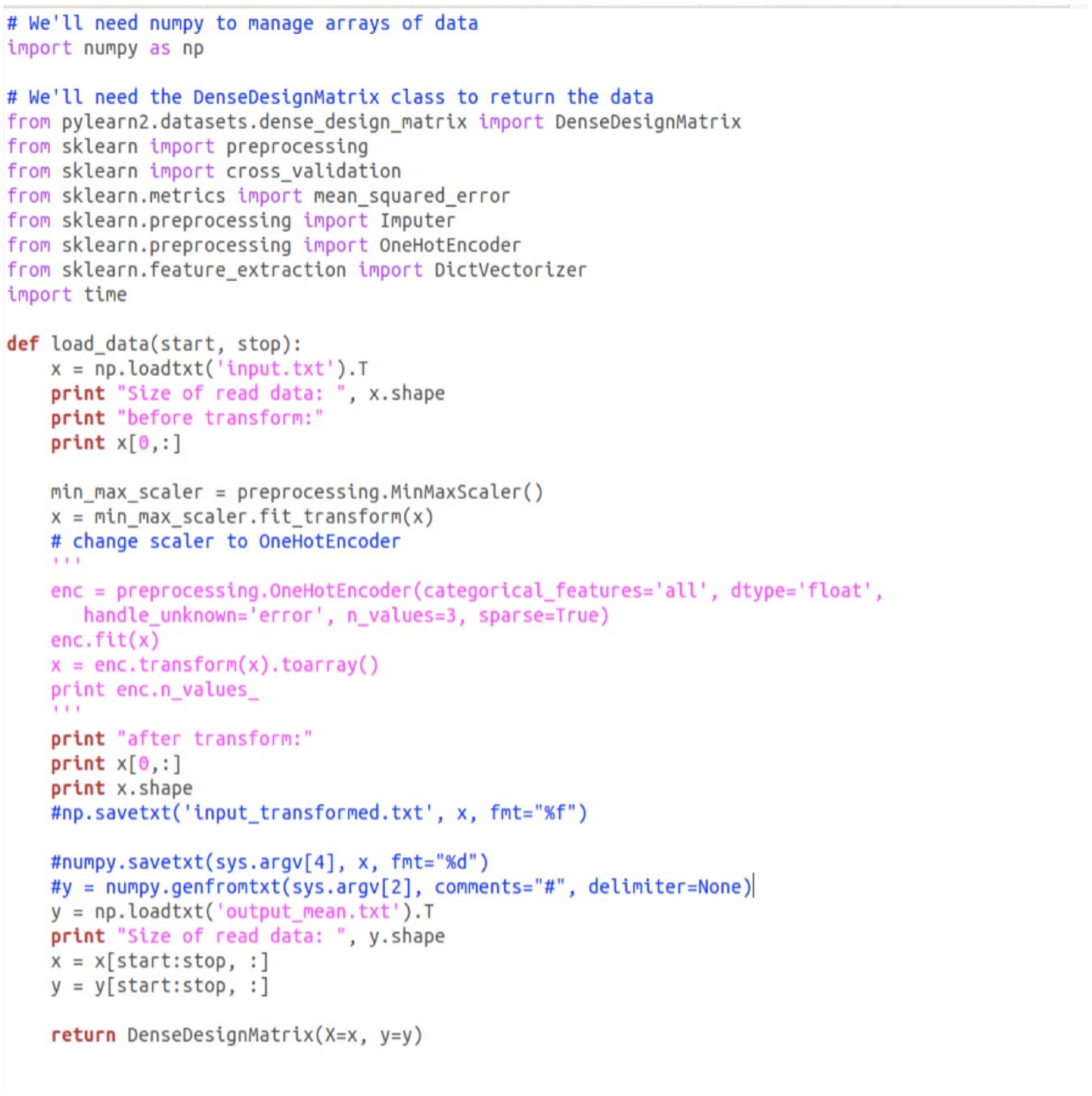
Figure A.4: The YAML file generated for the MLP model

A.3 Software Package

To use the software package, we need to use several commands for data processing, training and predicting.

A.3.1 Input and Output

In our demo folder, we need to preprocess the features in an input file, as illustrated in Figure A.5.



```
# We'll need numpy to manage arrays of data
import numpy as np

# We'll need the DenseDesignMatrix class to return the data
from pylearn2.datasets.dense_design_matrix import DenseDesignMatrix
from sklearn import preprocessing
from sklearn import cross_validation
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import Imputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.feature_extraction import DictVectorizer
import time

def load_data(start, stop):
    x = np.loadtxt('input.txt').T
    print "Size of read data: ", x.shape
    print "before transform:"
    print x[0,:]

    min_max_scaler = preprocessing.MinMaxScaler()
    x = min_max_scaler.fit_transform(x)
    # change scaler to OneHotEncoder
    '''
    enc = preprocessing.OneHotEncoder(categorical_features='all', dtype='float',
       handle_unknown='error', n_values=3, sparse=True)
    enc.fit(x)
    x = enc.transform(x).toarray()
    print enc.n_values_
    '''
    print "after transform:"
    print x[0,:]
    print x.shape
    #np.savetxt('input_transformed.txt', x, fmt="%f")

    #numpy.savetxt(sys.argv[4], x, fmt="%d")
    #y = numpy.genfromtxt(sys.argv[2], comments="#", delimiter=None)
    y = np.loadtxt('output_mean.txt').T
    print "Size of read data: ", y.shape
    x = x[start:stop, :]
    y = y[start:stop, :]

    return DenseDesignMatrix(X=x, y=y)
```

Figure A.5: The data process file generated for MLP model

A.3.2 Training Model

To start training a model, we need to use following python command.

```
$ python train_layers.py
```

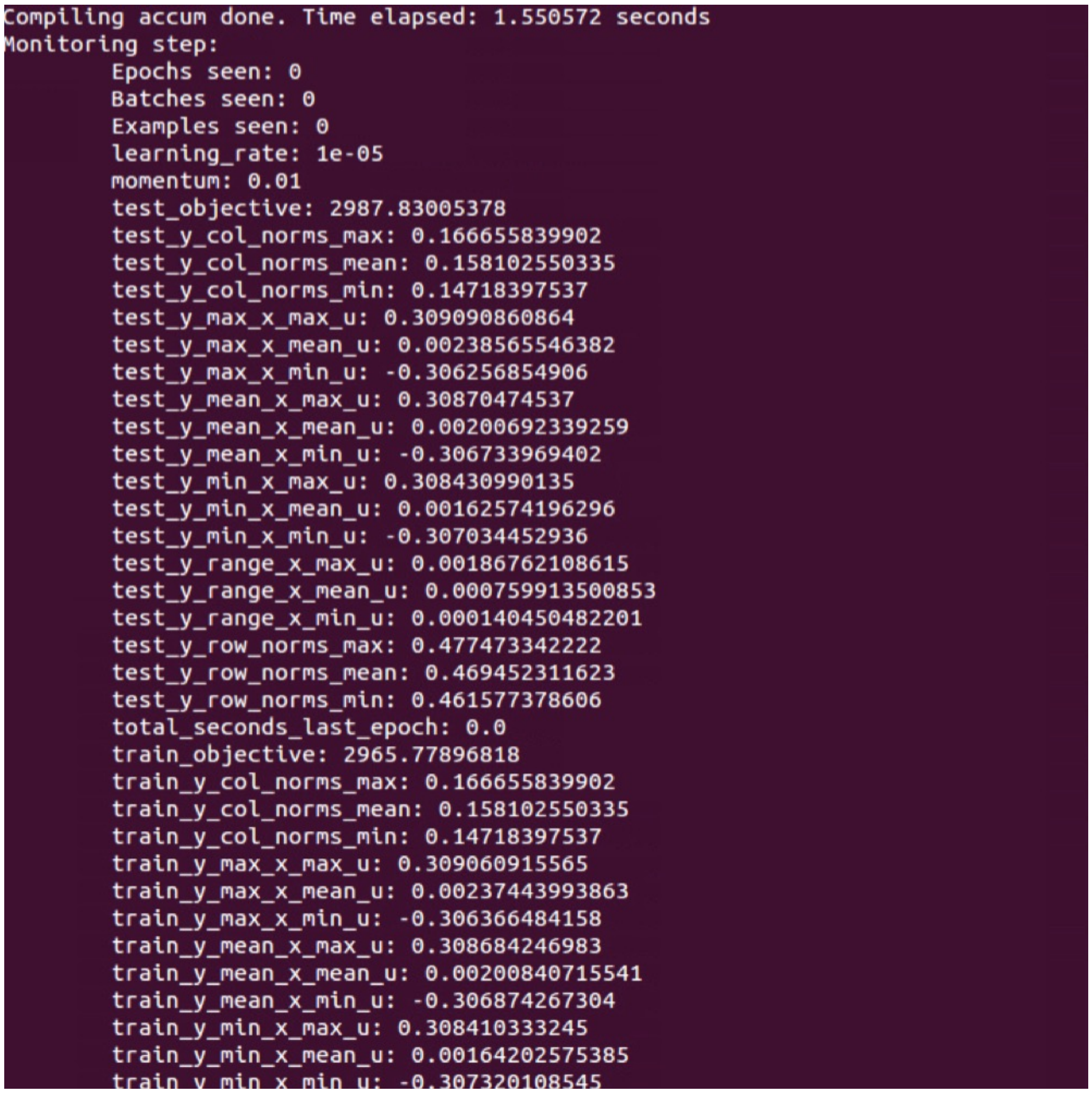
After training, we will get three *pkl* files storing the model for two autoencoder layers and an mlp layer. If we only need to train the mlp layer, we can just load to two *pkl* files of autoencoder layer generated before with following command.

```
$ python train_mlp.py
```

A.3.3 Monitoring Model

Pylearn2 provides a script that allows us to monitor model during training period and after training period.

During training, we are able to monitor the objective, and the script will save the model that minimizes our objective. Figure A.6 shows the monitor during training.



```
Compiling accum done. Time elapsed: 1.550572 seconds
Monitoring step:
        Epochs seen: 0
        Batches seen: 0
        Examples seen: 0
        learning_rate: 1e-05
        momentum: 0.01
        test_objective: 2987.83005378
        test_y_col_norms_max: 0.166655839902
        test_y_col_norms_mean: 0.158102550335
        test_y_col_norms_min: 0.14718397537
        test_y_max_x_max_u: 0.309090860864
        test_y_max_x_mean_u: 0.00238565546382
        test_y_max_x_min_u: -0.306256854906
        test_y_mean_x_max_u: 0.30870474537
        test_y_mean_x_mean_u: 0.00200692339259
        test_y_mean_x_min_u: -0.306733969402
        test_y_min_x_max_u: 0.308430990135
        test_y_min_x_mean_u: 0.00162574196296
        test_y_min_x_min_u: -0.307034452936
        test_y_range_x_max_u: 0.00186762108615
        test_y_range_x_mean_u: 0.000759913500853
        test_y_range_x_min_u: 0.000140450482201
        test_y_row_norms_max: 0.477473342222
        test_y_row_norms_mean: 0.469452311623
        test_y_row_norms_min: 0.461577378606
        total_seconds_last_epoch: 0.0
        train_objective: 2965.77896818
        train_y_col_norms_max: 0.166655839902
        train_y_col_norms_mean: 0.158102550335
        train_y_col_norms_min: 0.14718397537
        train_y_max_x_max_u: 0.309060915565
        train_y_max_x_mean_u: 0.00237443993863
        train_y_max_x_min_u: -0.306366484158
        train_y_mean_x_max_u: 0.308684246983
        train_y_mean_x_mean_u: 0.00200840715541
        train_y_mean_x_min_u: -0.306874267304
        train_y_min_x_max_u: 0.308410333245
        train_y_min_x_mean_u: 0.00164202575385
        train_y_min_x_min_u: -0.307320108545
```

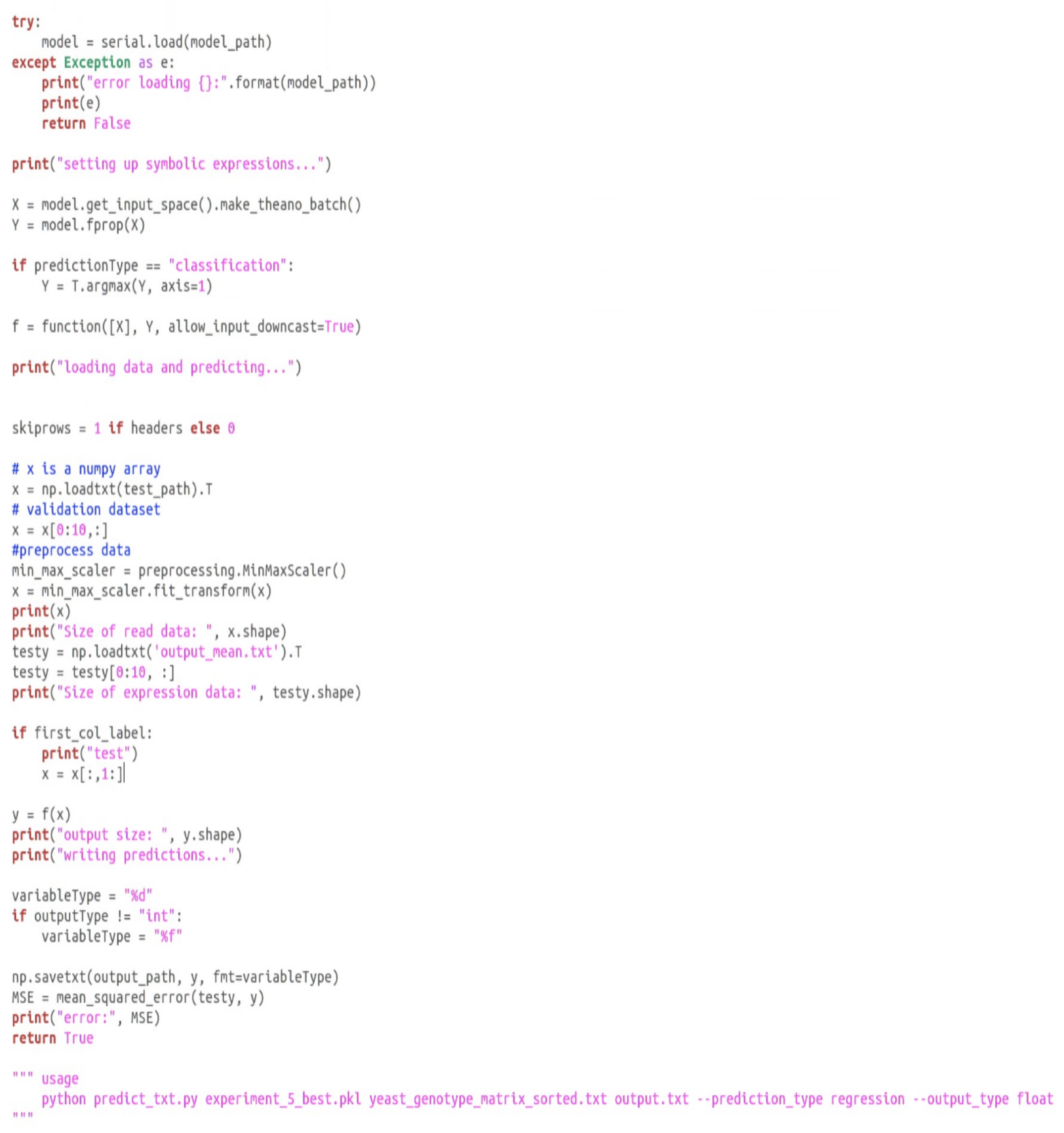
Figure A.6: Screen Monitoring

We can monitor the model after training with the following command example.

```
$python print_monitor.py experiment_6_best.pkl
```

A.3.4 Making Prediction

Figure A.7 shows part of code in prediction file.



```
try:
    model = serial.load(model_path)
except Exception as e:
    print("error loading {}:".format(model_path))
    print(e)
    return False

print("setting up symbolic expressions...")

X = model.get_input_space().make_theano_batch()
Y = model.fprop(X)

if predictionType == "classification":
    Y = T.argmax(Y, axis=1)

f = function([X], Y, allow_input_downcast=True)

print("loading data and predicting...")


skiprows = 1 if headers else 0

# x is a numpy array
x = np.loadtxt(test_path).T
# validation dataset
x = x[0:10,:]
#preprocess data
min_max_scaler = preprocessing.MinMaxScaler()
x = min_max_scaler.fit_transform(x)
print(x)
print("Size of read data: ", x.shape)
testy = np.loadtxt('output_mean.txt').T
testy = testy[0:10, :]
print("Size of expression data: ", testy.shape)

if first_col_label:
    print("test")
    x = x[:,1:]

y = f(x)
print("output size: ", y.shape)
print("writing predictions...")

variableType = "%d"
if outputType != "int":
    variableType = "%f"

np.savetxt(output_path, y, fmt=variableType)
MSE = mean_squared_error(testy, y)
print("error:", MSE)
return True

""" usage
    python predict_txt.py experiment_5_best.pkl yeast_genotype_matrix_sorted.txt output.txt --prediction_type regression --output_type float
"""
```

Figure A.7: Part of code in prediction file

We can make prediction after training with following command:

```
$ python predict_txt.py dea_mlp_best.pkl
$ input.txt output.txt --prediction_type
$ regression --output_type float
```

An output text file will be generated after the prediction.

Bibliography

[1] Seunghak Lee and Eric P Xing. Leveraging input and output structures for joint mapping of epistatic and marginal eqtls. *Bioinformatics*, 28(12):i137–i146, 2012.

[2] Xiaohui Chen, Xinghua Shi, Xing Xu, Zhiyong Wang, Ryan Mills, Charles Lee, and Jinbo Xu. A two-graph guided multi-task lasso approach for eqtl mapping. In *International Conference on Artificial Intelligence and Statistics*, pages 208–217, 2012.

[3] Wei Cheng, Xiang Zhang, Zhishan Guo, Yu Shi, and Wei Wang. Graph-regularized dual lasso for robust eqtl mapping. *Bioinformatics*, 30(12):i139–i148, 2014.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[5] Gautam V Pendse. A tutorial on the lasso and the shooting algorithm. 2011.

[6] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[7] Leo Breiman et al. Arcing classifier (with discussion and a rejoinder by the author). *The annals of statistics*, 26(3):801–849, 1998.

[8] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[9] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[10] Andy Liaw and Matthew Wiener. Classification and regression by randomforest.

[11] Richard Lewontin. The genotype/phenotype distinction. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2011 edition, 2011.

[12] Ohad Nachtomy, Ayelet Shavit, and Zohar Yakhini. Gene expression and the concept of the phenotype. *Studies in History and Philosophy of Science Part C: Studies in History and Philosophy of Biological and Biomedical Sciences*, 38(1):238–254, 2007.

[13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[14] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[15] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[16] Andrew Ng. Sparse autoencoder. *CS294A Lecture notes*, 72, 2011.

[17] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[18] Mark Richardson. Principal component analysis. 2009.

[19] Hervé Bourlard and Yves Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, 59(4-5):291–294, 1988.

[20] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[21] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408, 2010.

[22] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection.

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[24] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[25] Geoffrey Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9(1):926, 2010.

[26] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[27] I. J. Goodfellow, D. Warde-Farley, P. Lamblin, V. Dumoulin, M. Mirza, R. Pascanu, J. Bergstra, F. Bastien, and Y. Bengio. Pylearn2: a machine learning research library. *ArXiv e-prints*, August 2013.

[28] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[29] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua

Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

VITA

Rui Xie was born in the city of Heyuan, China. After finishing his undergraduate studies in 2014 from Northeastern University, China, majoring in Software Engineering, he started his graduate career at the University of Missouri, Columbia (MU).

In Fall 2013, Rui joined the MU Department of Computer Science in University of Missouri Columbia to pursue his M.S. studies under the advisory supervision of Professor Jianlin Cheng. His research is focused on applying deep learning to predict phenotypes from genotypes.