APPLICATION OF MACHINE LEARNING TECHNIQUES ON PREDICTION OF FUTURE PROCESSOR PERFORMANCE

by

Göktuğ İnal

Submitted to Graduate School of Natural and Applied Sciences

in Partial Fulfillment of the Requirements

for the Degree of Master of Science in

Computer Engineering

Yeditepe University

2018

APPLICATION OF MACHINE LEARNING TECHNIQUES ON PREDICTION OF FUTURE PROCESSOR PERFORMANCE

APPROVED BY:

Assoc. Prof. Dr. Gürhan Küçük (Thesis Supervisor)	[illegible]
Prof. Dr. Sezer Gören Uğurdağ	[illegible]
Prof. Dr. Yusuf Sinan Akgül	[illegible]

DATE OF APPROVAL: ..../..../2018

ACKNOWLEDGEMENTS

I would like to thank my advisor Professor Gürhan Küçük for his extraordinary support in this thesis process. Pursuing my thesis under his supervision has been an experience, which broadens the mind and presents an unlimited source of learning. This thesis would have been impossible without the patience and the support of my friends.

Finally, I would like to thank my family for their endless love and support, which makes everything more beautiful and easy.

ABSTRACT

APPLICATION OF MACHINE LEARNING TECHNIQUES ON PREDICTION OF FUTURE PROCESSOR PERFORMANCE

Today, processors utilize many data path resources with various sizes. In this study, we focus on single thread microprocessors, and apply machine learning techniques to predict processors’ future performance trend by collecting and processing processor statistics. This type of a performance prediction can be useful for many ongoing computer architecture research topics. Today, these studies mostly rely on history- and threshold-based prediction schemes, which collect statistics and decide on new resource configurations depending on the results of those threshold conditions at runtime. The proposed offline training-based machine learning methodology is an orthogonal technique, which may further improve the prediction accuracy of such existing algorithms. We show that our neural network based prediction mechanism achieves around 70 per cent accuracy for prediction performance trend (gain or loss in the near future) of applications.

ÖZET

GELECEK İŞLEMCİ PERFORMANSININ TAHMİNİNDE MAKİNE ÖĞRENME TEKNİKLERİNİN UYGULANMASI

Günümüzde, işlemciler çeşitli boyutlarda birçok veriyolu kaynağını kullanmaktadır. Bu çalışmada, tek iş-parçacıklı mikroişlemciler üzerinde durmakta ve çalışan bir uygulamanın gelecekteki performans trendini tahmin etmek için makine öğrenme tekniğini uygulamaktayız. Bunu yaparken işlemci istatistiklerini toplamakta ve işlemekteyiz. Bu tür bir performans tahmini süregelen bilgisayar mimarisi araştırma konuları için de yararlı olacağını öngörmekteyiz. Bugün, bu çalışmalar çoğunlukla işlemci istatistikleri toplayan ve programın çalışma süresince eşik durumlarına bağlı olarak yeni kaynak konfigürasyonlarına karar veren, eşik tabanlı tahmin yöntemlerine dayanmaktadır. Önerilen çevrimdışı eğitim tabanlı makine öğrenme metodolojisi, mevcut algoritmalarının tahmin doğruluğunu daha da arttırabilecek ortogonal bir tekniktir. Yapay sinirsel ağ tabanlı tahmin mekanizmamız tahmini işlemci performans eğilimi (yakın gelecekte kazanç ya da kayıp) için yüzde 70 doğruluk oranına ulaşmaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

ALU	Arithmetic Logic Unit
ALUocc	Arithmetic Logic Unit occupancy
ARF	Architectural Register File
ANNs	Artificial Neural Networks
CPU	Central Processing Unit
, : à$_4$áà$_5$;	Cost function
DIPC	Dynamic Instructions Per Cycle
FQocc	Fetch Queue Occupancy
D	Hypothesis function
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
LSQ	Load/Store Queue
LSQocc	Load/Store Queue occupancy
IQ	Issue Queue
λ	Lambda
µ	Mean value,
PRF	Physical Register File
ROB	Re-Order Buffer
ROBocc	Re-Order Buffer occupancy
ê	Sigma value
g	Sigmoid function
θ	Theta
ALUtot	Total number of Arithmetic Logic Unit entries
FQtot	Total number of Fetch Queue entries
LSQtot	Total number of Load/Store Queue entries
ROBtot	Total number of Re-Order Buffer entries
: ^{í}	Transpose of X matrix

1. INTRODUCTION

Today, computer architecture research focuses on prediction mechanisms for adaptive and reconfigurable components that are resized and/or restructured according to runtime requirements of applications. This kind of adaptation might be especially helpful for improving the energy savings that are realizable on a given system. When, an application does not require all the data- and control-path structures within a processor, unused portions of those structures might be especially helpful for improving the energy savings that are realizable on a given system. When, an application does not require all the data- and control-path structures within a processor, unused portions of those structures might be gated off, resulting in energy and power savings on the entire system. However, a microprocessor consists of many structures that either directly or indirectly effect the overall system performance, and, the worst of all, not all of these structures have an effect on the performance at all times. Sometimes, the Issue Queue (IQ) may be full, and it might be the structure that is responsible for stalling the front-end of the processor pipeline and degrading performance. Sometimes, the L1 cache may take the full responsibility for such performance drop, since the working set of a running application may be quite large to fit into such small cache. In other occasions, the branch predictor, the Re-Order Buffer (ROB), the Load/Store Queue (LSQ), L2 shared cache and many other data path structures can play a role on processor performance drop, solitary or altogether. As a result, nondeterministic nature of a microprocessor makes accurate prediction of performance a hard problem, which must be attacked very wisely.

Today, the main strategy to accurately predict the future performance of a running application is to rely on history- and threshold-based algorithms. For instance, to predict the future resource requirements of an application, its most recent indicators for resource usage are collected and analyzed at runtime. When those indicators are above a certain, empirically decided, threshold, then a decision algorithm may predict that the same behavior might be observed in the near future, as well. However, such prediction mechanisms have two major weaknesses. First, they must be designed with both power and latency concerns, since they are required to periodically run on hardware. In our proposed design, since we train our prediction model offline, we can train it as long as and as much as we like, and we do not have such concerns. Secondly, since online prediction algorithms need to work on a limited window of historical data, they may fail seeing the big picture. For instance, when an application shows an oscillating behavior, their prediction may almost always miss its target. In our proposed offline learning method, we can analyze any amount of data in any window size, and this gives us greater flexibility for generating prediction functions with a higher accuracy, as long as we train it well. However, due to its offline nature, our method has its own limitations especially originating from its training mechanism. Today, we see that processor companies are also in the discovery phase of this new research path. For instance, AMD’s the most recent processor Ryzen is utilizing a neural network predictor for improving the accuracy of its branch prediction mechanism [1].

Since last decade, the machine learning algorithms are proven to be useful in many computer science domains. In computer architecture research, there are a variety of studies, which may get benefit from these learning techniques. Our main motivation in this study is to accurately predict the future performance of applications by offline analysis and training using regression models and Artificial Neural Networks (ANNs). There are only a few studies in the literature targeting similar topics. However, our proposed study is a unique one, which targets better accuracy on prediction of processor performance. If this prediction is made accurate enough, then behavior of running applications can be better tracked, and, thus, processor resources can be better utilized, resulting in higher energy savings. Here, we show that application of regression models and ANNs for supervised learning is a promising land for this research domain. As a result, we believe that offline training has its merits compared to existing prediction models that depend on mere runtime statistics, and it might help us further improve the prediction accuracy of processor performance.

2. RELATED WORK

The most related study to our work builds linear regression models to relate processor performance to micro-architectural parameters for design space exploration [2]. Similarly, the authors use a detailed cycle-accurate superscalar processor simulator to collect runtime statistics. However, the main motivation of the study is quite different. They try to predict processor performance for any given micro-architectural parameters. In contrast, here, we focus on predicting applications' performance for a fixed processor configuration. Our major goal is to provide highly accurate performance predictions to algorithms with various motivations. For instance, a resource partitioning algorithm can make use of such accurate predictions for better resource utilization and higher power savings. There are many other motivating studies that can make use of such precious information [3][4][5][6][7].

Another similar study predicts processor performance by building empirical functions that integrate micro-architectural parameters for a typical superscalar processor [8]. Again, the main goal is faster design space exploration as in [2]. This earlier study aims to accurately predict the Instructions Per Cycle (IPC) at the end of the completion of an application. The authors claim that the predicted IPC is within 5.8 per cent range of the actual IPC, on the average. Though, in this study, our main motivation is to accurately predict IPC values in certain time periods in the near future, so that an existing algorithm can benefit from such prior knowledge adapting itself to either an unavoidable performance drop or a performance increase. Besides, instead of building hand-made empirical functions, we let machine learning techniques to discover such functions by applying regression models.

Finally, there is a study on performance prediction for parallel applications, which is similar to our neural network approach [10]. The authors consider that it is difficult to construct analytical predictive models even though they are useful. Similarly, they build multi-layer neural networks trained on input data, that is, there is a hidden layer, which transforms a single-layer structure into multi-layer structure. However, in our study, we focus on single thread of M-Sim instead of parallel applications SMG2000. They gather performance samples from application, which is executed and use those data in their constructed training neural network system. The author claims that some studies on SMG2000 have been carried out, but the code's variations in execution time are not well understood [10]. Thus, they focused on avoiding noise in their data set and collecting appropriate sampling techniques.

However, our main motivation is to accurately predict trend of IPC in the near future, so that existing algorithms can adapt itself to unpredictable increase or decrease of processor performance.

3. MACHINE LEARNING MODELS

3.1. REGRESSION MODELS

A regression model is represented by a mathematical formulation, which allows us to accurately predict an output for any given input. In this technique, the relation between the inputs and their corresponding output is investigated by the help of a training set of inputs and outputs. At the end of this supervised learning process, a formulation for the given problem is generated.

The formulation of *h* function, which stands for hypothesis, is given in Eq.3.1 (linear regression). Ideally, θ_0 and θ_1 are chosen so that *h(x)*, result of the value, which is predicted on input x, is close to the actual output value obtained from the training data set.

$$h(x) = \theta_0 + \theta_1 x \tag{3.1}$$

In some cases, there might be many input parameters that can affect the estimated value. In such a case, the straightforward model in Eq.3.1 may not be sufficient, and a more general formula is used, as shown in Eq.3.2 below.

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n \tag{3.2}$$

In a regression problem, the cost minimization formula is commonly used to obtain an accurate prediction result. The mathematical definition of a cost function, J, is the squared error function as shown in Eq.3.3, and it is measured each iteration until it goes below a certain threshold value.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h\left(x^{(i)}\right) - y^{(i)} \right)^2 \tag{3.3}$$

Here, m represents the number of training samples and y represents the set of actual outputs from the training data set. When the cost function converges, the overall objective function of a linear regression is obtained.

The linear regression model can be developed in many ways. The features or input values can be combined into one variable, or a new feature can be created by just having square of each input. The forms of polynomial regression models called quadratic regression and cubic regression are shown in Eq.3.4 and Eq.3.5, respectively.

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_1 x_1^2 + \theta_2 x_2 + \theta_2 x_2^2 + \dots + \theta_n x_n + \theta_n x_n^2 \qquad (3.4)$$

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_1 x_1^2 + \theta_1 x_1^3 + \theta_2 x_2 + \theta_2 x_2^2 + \theta_2 x_2^3 + \dots + \theta_n x_n + \theta_n x_n^2 + \theta_n x_n^3 \qquad (3.5)$$

Our goal in this study is to accurately predict the trend in processor performance in terms of the IPC in periodic time intervals. Since, the performance metric represents an application's dynamic performance behavior within each time period, we call it the Dynamic Instructions Per Cycle (DIPC), instead. It is essential to observe how far off the estimated the DIPC value from the actual DIPC, and, for that purpose, we consider three different ranges (3 per cent, 5 per cent and 10 per cent) throughout this study. We also study linear, quadratic and cubic regression models, and as a by-product, we obtain minimized weights (theta values) and, hence, accurate prediction trends for processor performance.

3.2. BUILDING A LINEAR MODEL

We limit our approach as implementing a linear, quadratic and cubic regression models to predict the performance of SPEC CPU2006 benchmarks in terms of the DIPC values. In this study, we restrict the number of architectural parameters to only four: 1) L1 cache miss rate, 2) L2 cache miss rate, 3) measured level of Instruction Level Parallelism (ILP), and 4) average number of dynamic instructions per cycle. Thus, we collect these processor statistics for a period of time, and, then, we use them to predict the next DIPC at the end of a consequent period. During our experiments, we minimize the cost function given in Eq.3.3 and use the resulting function for our performance prediction.

We take the hypothesis function in Eq.3.1, and we attempt to measure how well it fits into our training data. Thus, we need to predict both $à_4$ and $à_5$. Here, a gradient descent function, which is shown in Eq.3.6, is used.

$$à_Ý \; L \; à_Ý \; F \; Ù\frac{!}{! \; õ}, : à_4 á à_5 ; \qquad (3.6)$$

In Eq.3.6, j represents the feature index and each iteration all theta parameters $à_5 á à_6 á å \; á à_á$ are simultaneously updated. We select α parameter carefully to ensure that the gradient descent algorithm converges. Note that, if α value is too small, the algorithm can converge after a long run. In other cases, it may fail to converge or it may even diverge.

There is another way of estimating the theta parameter. This method is mostly used for small data with no iteration, as shown in Eq.3.7.

$$à \; = :: \, ^{Í} : \, ;^{?5} : \, ^{Í} U \qquad (3.7)$$

Here, X is our training data matrix and $:^{Í}$ X is invertible. In addition, normalization can be applied for some feature sets. As the result of the normalization, those features are scaled to have two parameters: ê and ä. The mathematical formula is given in Eq.3.8.

$$: \; L \; \frac{Ñ?}{\;} \qquad (3.8)$$

In this equation, ä is the mean value and ê is the standard deviation of the feature data set. The cost function is calculated by applying the gradient descent, the normal equation and normalization algorithms. Theta parameters are simultaneously updated. The accuracy of our estimation depends on the size and the entropy of the training data set, the alpha parameter and, of course, the number of iterations.

Algorithm 3.1 given below describes the details of the gradient descent method. In the initial phase of the algorithm, features are normalized and gradient descent model is applied to our training data. The alpha and the maximum number of iterations are set. Then, the gradient descent function is called to compute the theta parameter. At the 9th line of the pseudo code, J value is calculated with a formula represented in Eq.3.3. Finally, at line 10, we iteratively calculate a new theta value according to the function given in Eq.3.6.

Algorithm 3.1. Pseudo code of the gradient descent

```
 1: Initialize-parameters
 2: X ← training data, y ← testing data
 3: X ← featureNormalization(X)
 4: alpha ← 0.1, 0.01 or 0.001
 5: maxIters ← 50
 6: theta ← 0
 7: m ← length of y
 8: for ( i ← 0, i < maxIters, i++) do
 9:   J ← 1/(2 * m)* 2 *( $X^T$ * X * theta - $X^T$ * y)
10:   theta ← theta - alpha * J
11: end for
```

3.3. NEURAL NETWORK MODEL

A Neural Network (NN) model enables a computer to learn from a set of training data as in the regression models. They are used to solve a large variety of machine learning problems. Regression models have already been used and accurate results were obtained. Thus, why we need ANNs learning? At Fig.3.1, supervised learning classification problem is represented with a training data set. It is called supervised learning because it is the process of teaching from a training data set. When the learning process reaches an acceptable level, learning stops. In the figure, the method works well only if there are two features x_1 and x_2. However, many machine learning problems have a lot more features than just two.

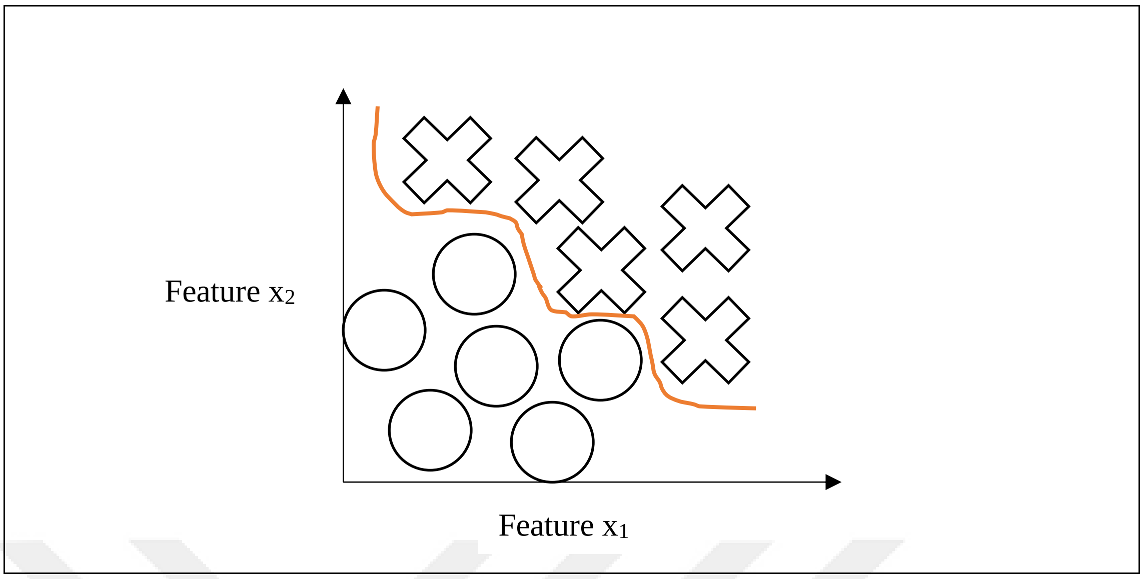

Figure 3.1. Classification of two different features

Considering the same figure, let’s say, there are 100 different features, n = 100. When polynomial regression is applied to those features, there would be numerous computations or multiplications which generate about 5000 quadratic or polynomial terms :• ^{6};. Thus, the size of terms might cause overfitting the training set and also it is a computationally expensive process. On the other hand, if the cubic model is set, there would be a huge number of features :• ^{7};. When n is large, there is not a proper way to build regression models for many machine learning problems.

When a dog image is considered in Fig. 3.2, machine learning can be used to train a classifier to examine that image and reasonable outcomes might be obtained whether or not the image is a dog. There are numerous pixel values that these numbers represent the nose of the dog, for example.

Figure 3.2. The image of a dog

In this process, a machine learning technique is generated to detect a dog image. A few label examples of dogs and a few examples of not dogs are collected. The data are given to learning algorithm in order to be trained and predictions are made on the test data. However, suppose that each image has 50x50 pixel, that would be 2500 pixels (i.e. n = 2500). If system attempts to learn the non-linear hypothesis by including all quadratic features, that means there would be almost 3 million features, and that would be quite expensive (in terms of computational cost) to represent all these features per training set. Thus, linear, quadratic, or cubic regression models are not good methods to learn complex non-linear hypothesis when n is too large.

Fig.3.3 shows the illustration of neural network model. At a very simple level, neurons are basically computational units and they take inputs (dendrites). Those dendrites are channeled to outputs (axons). In our model, our dendrites represent the input features $š_5$ å $š_1$ and the # “bias unit” which is always equal to 1 is determined as input node. In an artificial neural network, sigmoid (logistic) function is used as in the classification process, and theta parameters are called “weights.”

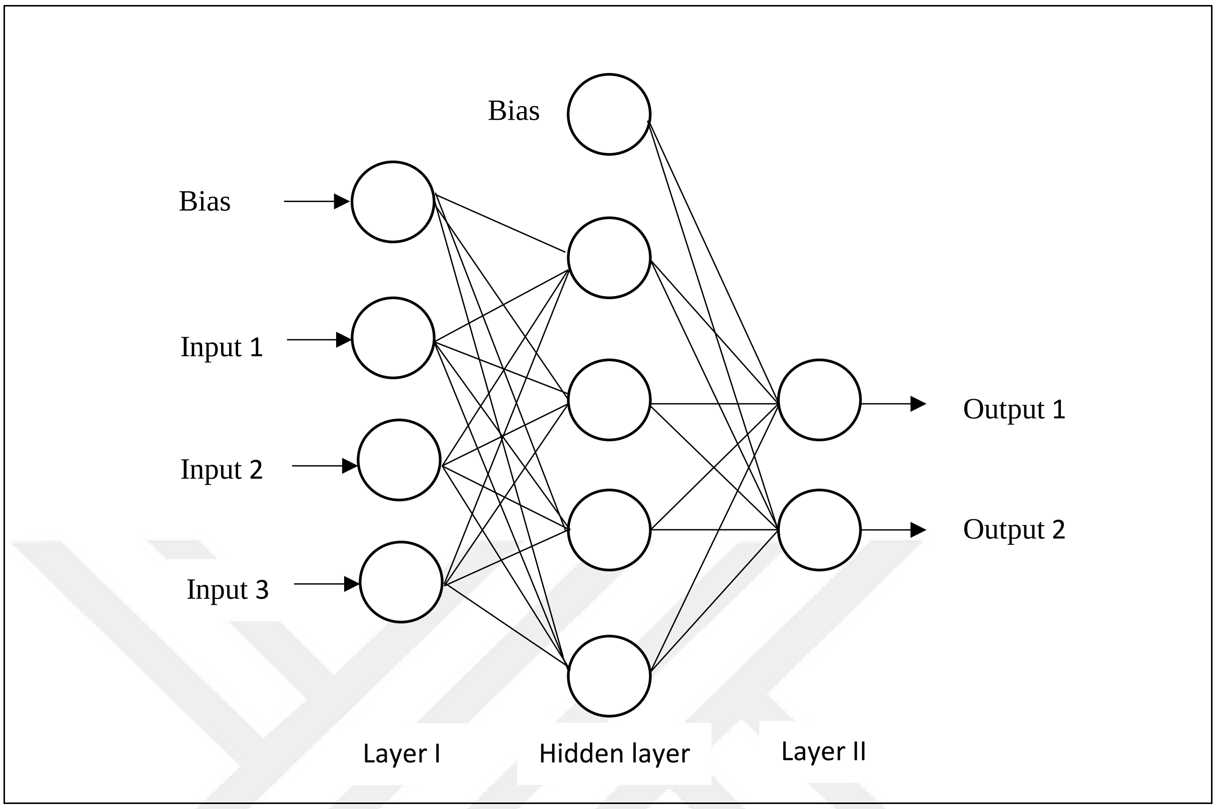

Figure 3.3. Illustration of the neural network model

In Fig.3.3, layer I represents the input layer, which works on training and testing data. The input layer is connected to the output layer, which outputs the hypothesis function, via a hidden layer as also shown in Eq.3.9.

$$f\begin{matrix}š_4\\š_5\\š_6\\š_7\end{matrix}j \setminus \quad f\begin{matrix}=^{:6;}_5\\=^{:6;}_6\\=^{:6;}_7\end{matrix}j \setminus \quad Š_€:š; \tag{3.9}$$

The symbols of $=^{:h;}_{g}$ and $\#^{:h;}$ represent hidden unit i in layer j and matrix of weight from layer to layer i + 1, respectively. The values of each of hidden layer nodes are calculated as shown in equations 3.10, 3.11, 3.12 and 3.13.

$$=^6_5 \; L \; \%@\#^{:5;}_{54}š_4 \; E \; \#^{:5;}_{55}š_5 \; E \; \#^{:5;}_{56}š_6 \; E \; \#^{:5;}_{57}š_7A \tag{3.10}$$

$$=^6_6 \; L \; \%@\#^{:5;}_{64}š_4 \; E \; \#^{:5;}_{65}š_5 \; E \; \#^{:5;}_{66}š_6 \; E \; \#^{:5;}_{67}š_7A \tag{3.11}$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3) \quad (3.12)$$

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}) \quad (3.13)$$

According to the NN model, each hidden node is computed by using an n x m matrix of parameters. To obtain one hidden node, each row of the parameters is applied to our input or training data sets. Our hypothesis output is a sigmoid function (g) that applied to the sum of the values of our hidden layer nodes, which are multiplied by another parameter matrix Θ. Thus, weights are obtained for second layer of nodes and each layer has own matrix of weights Θ.

Vectored implementation is applied by considering previous equations. Parameter z is encompassed inside our g (sigmoid) function in Eq.3.14, 3.15 and 3.16. If variable $z_k^{(j)}$ is replaced for all parameters.

$$a_1^{(2)} = g(z_1^{(2)}) \quad (3.14)$$

$$a_2^{(2)} = g(z_2^{(2)}) \quad (3.15)$$

$$a_3^{(2)} = g(z_3^{(2)}) \quad (3.16)$$

In other words, for layer j = 2 and node k, the parameter z will be:

$$z_k^{(2)} = \Theta_{k,0}^{(1)}x_0 + \Theta_{k,1}^{(1)}x_1 + \cdots + \Theta_{k,n}^{(1)}x_n \quad (3.17)$$

And the vector representation of x and $z^{(j)}$ is:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \quad z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \vdots \\ z_n^{(j)} \end{bmatrix} \quad (3.18)$$

When x is set to $a^{(1)}$, the equation is written as:

$$z^{(j)} = \Theta^{(j-1)}a^{(j-1)} \quad (3.19)$$

Matrix $\Theta^{(j-1)}$ has $s_j \times (n+1)$ dimensions (where s_j is the number of our hidden layer) by our vector $a^{(j-1)}$ with height $(n+1)$. This creates vector $z^{(j)}$ with height s_j. Thus, our hidden layer function is represented for layer j in equation:

$$a^{(j)} = g(z^{(j)}) \tag{3.20}$$

After $a^{(j)}$ is calculated, bias unit can be added to layer j. Bias unit is represented by $a_0^{(j)}$ and it will be equal to 1. Thus, last version of our z function is:

$$z^{(j+1)} = \Theta^{(j)} a^{(j)} \tag{3.21}$$

In this equation, $\Theta^{(j)}$ has only one row so that it is multiplied by one column $a^{(j)}$ and the result, which is obtained, is a single number. The final result is illustrated in equation:

$$h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)}) \tag{3.22}$$

Adding this intermediate layer to our NN model enables us to build more complex non-linear hypotheses.

We need to define a few parameters before illustrating cost function equation:

- L=total number of layers
- s_l=number of units in layer l (disregard bias unit)
- K=number of output units

In Eq.3.23, a more complicated neural network cost function with regularization is given.

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log\left(\left(h_\Theta(x^{(i)})\right)_k\right) + \left(1-y_k^{(i)}\right)\log\left(1-\left(h_\Theta(x^{(i)})\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{j,i}^{(l)}\right)^2 \tag{3.23}$$

In the regularization part, after the square brackets, multiple theta matrices must be explained. The number of columns in our theta matrix is exactly same as the number of nodes in the next layer (bias unit is including). The number of rows in our theta matrix is exactly same as the number of nodes in the next layer (bias unit is not including). In addition, there are additional nested summation loops through the number of output nodes before the square brackets in the first part of equation.

If a non-multiclass classification is performed (k=1) and if we exclude regularization, cost function becomes

Ⱡ. :–; L ›$^{:r;}$ Ž‰o @Š$_{€}$kš$^{:g}$ oAE ks F ›$^{:r;}$ oŽ‰o @ F Š$_{€}$kš$^{:r;}$ oA (3.24)

Weights (Θ) must be initialized randomly before starting a neural network process. If those weights are set to zero, the neural network would not work properly. Thus, all nodes would update to the same value, repeatedly. To avoid such a problem, weights can be initialized randomly using the following method.

- Initialize each $\Theta_{ij}^{(l)}$ to a random value in [-ϵ,ϵ]
- Theta1=rand(5,6)*(2*INIT$_{\text{EPSILON}}$)-INIT$_{\text{EPSILON}}$
- Theta2=rand(1,6)*(2*INIT$_{\text{EPSILON}}$)-INIT$_{\text{EPSILON}}$

Therefore, $\#^{:j;}_{gh}$ is set to a random value between [-epsilon, epsilon]. Same operation is implemented to all the Θ's. In this method, rand(x, y) is just a ready function in octave environment. Thanks to this function, random values between 0 and 1 can be obtained.

In a neural network process, sets of matrices are used and it is necessary to put them into one long vector in order to use optimizing function such as *fminunc*.

Then, original positions of matrices must be provided from unrolled version as follows:

- Theta1 = reshape(thetaVector())
- Theta2 = reshape(thetaVector())
- Theta3 = reshape(thetaVector())

The neural network architecture is picked, layout of it is decided, and the number of hidden units in each layer and in total is determined. Then, some adjustments are applied such as the number of input units which are the dimension of features š:g, the number of output units which stands for the number of classes, the number of hidden units which is better if it is more and it is recommended that if there are more than 1 hidden layer, it is appropriate to select same number of units in every hidden layer.

The phases of a training neural network are ordered as follows:

- Randomly initialize weights
- Forward-propagation to calculate Š€:š:g;
- Cost function
- Back-propagation to compute partial derivatives
- Gradient checking
- Built-in predict functions to make an accurate calculation

These six phases are repeated for every training example. In the end, the desirable result would be Š€:š:g; ≈ . Thus, cost function would gradually approach to a minimum point. However, note that the result can end up in a local minima.

Algorithm 3.2. Pseudo code of NN predict function

```
1: predict(Theta1, Theta2, X)
2: Initialize-parameter
3: h1 = sigmoid([ones(m, 1) X] * Theta1')
4: h2 = sigmoid([ones(m, 1) h1] * Theta2')
5: [dummy, p] = max(h2, [], 2)
6: return p
7: end
```

The pseudo code in Algorithm 3.2 returns prediction the label of an input given a trained neural network. The function outputs the credited label of X given the trained weights of natural network (Theta1, Theta2). At 3rd and 4th lines of the pseudo code, the sigmoid function computes the sigmoid of z so that inputs are matched to outputs. Most sigmoid functions positively derive, continuous and they are straightforward to calculate. It is similar to a black box that takes an input and produces an output. The Max function returns index of the neuron with the highest value. It stores in p indices of h2, which have the highest values along dimension 2. In the algorithm, h2 has activations of each output neuron.

Algorithm 3.3. Pseudo code of prediction accuracy

```
 1: Initialize-parameter
 2: featureNormalize(X)
 3: Initialize-weights
 4: data = testing data
 5: trend = testing data of DIPC trend
 6: m = size of data
 7: success = 0
 8: for ( i = 1 : m) do
 9:   pred = predict(Theta1, Theta2, data(i, :))
10:   if ((trend(i)+1) == pred) do
11:       success = success + 1
12:   end if
13: end for
```

First, a few parameters are set to use for neural network accuracy prediction. Input layer size, hidden layer size, and number of labels, which represents output nodes, are determined in Algorithm 3.3. These parameters show diversity according to how many features are used in ANNs. Second, the featureNormalize function is called in order to return add normalized version of X or input variables. In the 3rd line of the Algorithm 3.3, weights of neural network are initialized randomly. A randomly initialized function is used to do so. After training neural network, testing data is used to predict the labels, which are determined by num_label parameter. In the 8th line of the Algorithm 3.3, for loop iterates all testing inputs one by one. By using the predict function, trend of DIPC is calculated and it is compared to the current trend data so that rate of the successful or accurate predictions can be statistically computed.

Algorithm 3.4. Pseudo code of feedforward

```
nnCostFunction(nnparameters, inputl_size, hiddenl_size, num_labels, X, y, lambda)
Initialize-parameters
Reshape-Theta1 and Theta2
J = 0
K = num_labels
for (i : m) do
    X_i = X(i : m)
    h_of_Xi = sigmoid ([1 sigmoid(X_i * Theta1')] * Theta2')
    y_i = zeros(1,K)
    y_i(y(i) + 1) = 1
    J = J + sum(-1 * y_i .* log(h_of_Xi - (1-y_i) .* log(1 - h_of_Xi)))
end for
J = 1/m *J
J = J + (lambda / (2*m) * (sum(sumsq(Theta1(:,2:inputl_size+1))) + sum(sumsq(Theta2(:,2:hiddenl_size+1)))))
return J
```

The NN cost function computes the cost and the gradient of the neural network. In the neural network structure, the parameters are unrolled vector of the partial derivatives of the neural network. Thus, those parameters are reshaped into Theta1 and Theta2, which are the weights of our two-layer neural network.

The Feedforward algorithm returns the cost variable J as an output. y_i is a vector contains 0's as much as the number of K. K represents number of output units. In our NN structure, we have two output units which shows increase or decrease trend (K = 2). Then, first or second slot of K vector is set to 1 according to output of y. If y(i) is equal to 0, then y_i = [1 0]. On the other hand, if y(i) is equal to 1, then y_i = [0 1]. By the help of y_i vector, computation of J minimum is carried out.

4. ARCHITECTURAL DESIGN

Instructions Per Cycle (IPC), which reflects computer performance, expresses the average number of instructions executed for each clock cycle. In our approach, we focus on Dynamic Instructions Per Cycle (DIPC). The DIPC represents the average number of instructions executed during a period, which we call epoch. Here, the epoch size is empirically set to one million clock cycles throughout this study. We tried shorter or longer durations for the epoch size, and shorter durations give us too many redundant data whereas longer durations do not successfully capture phase changes of running benchmark applications.

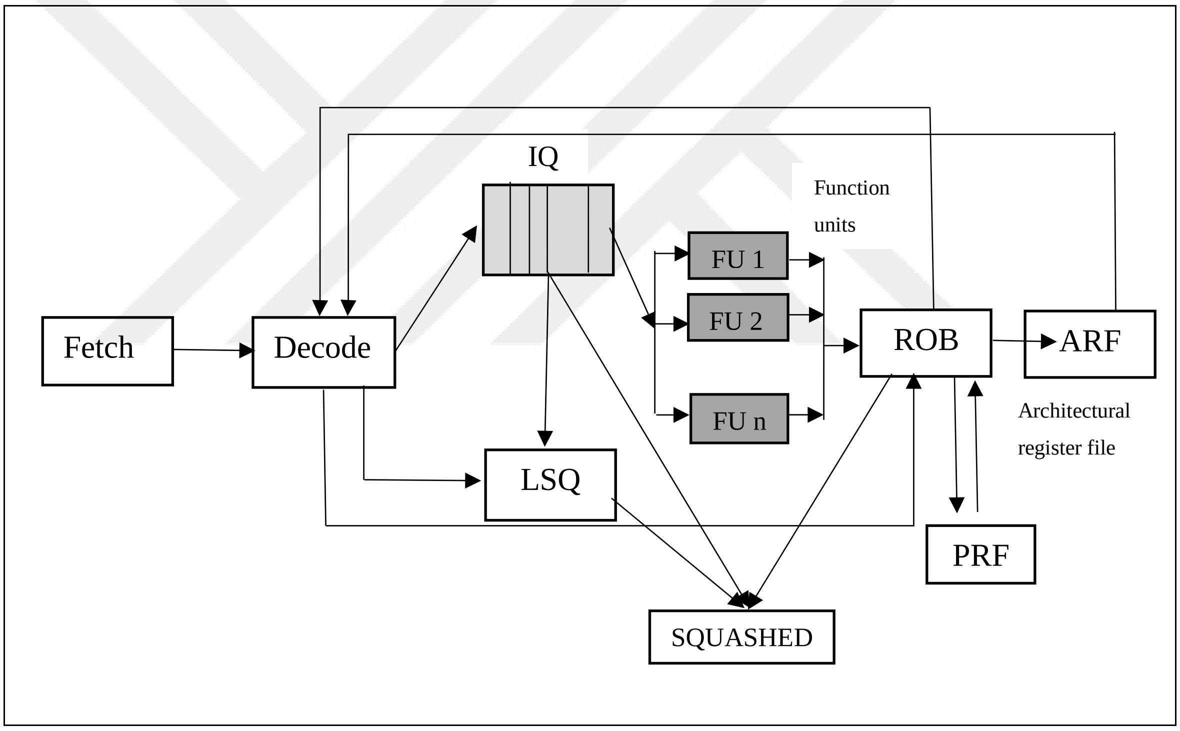

Figure 4. 4. Superscalar datapath

Fig.4.4 depicts the superscalar datapath that we considered for this study. Here, instructions are fetched and decode stage starts. Some instructions are located in Issue Queue (IQ) and load/store instructions are hold in Load/Store Queue (LSQ). Thus, instructions are stored to physical registers in function units. The Re-Order Buffer (ROB) entry set up for an instruction at the time of dispatch contains a field to hold the result produced by the instruction. A dispatched instruction attempts to read operands from the Architectural Register File (ARF) directly when the operand is committed. Otherwise, the operand is generated but not committed so that it is read from the ROB. The operand values are hold in physical register file (PRF) when they are not committed. If the operand values are no longer used, they are forwarded to squashed stage.

L1 and L2 data caches are used to reduce the average time to access memory. They are located within the processing unit of the computer. These two caches are directly related to performance of processor. Firstly, Central Processing Unit (CPU) starts to execute some instructions when a request or an access reaches to system. Then, CPU uses level 2 cache to cut down delay. L1 cache is too small in comparison to L2. It is the first destination for memory instructions. However, CPU looks in the level 2 cache, which has some latency and large area, when addresses are not found inside the level 1 cache. You can see the calculation of L1 and L2 caches below in Algorithm 4.5. When a request reaches to system, d1_access or dl2_access parameters are increased. If event is missed, dl1_miss or dl2_miss parameters are increased.

Algorithm 4.5. Pseudo code of the calculation of L1 and L2 caches

```
initialize-parameters                      ]
counter_t dl1_miss = 0                     ]
counter_t dl1_access = 0                   ]  At initialize
counter_t dl2_miss = 0                     ]
counter_t dl2_access = 0                   ]
if (cache_dl1) do                          ]
  if (lat > cache_dl1_lat)                 ]
      dl1_miss++                           ]
  end if                                   ]
  dl1_access++                             ]
end if                                     ]  Every cycle at decode
if (cache_dl2) do                          ]
  if (lat > cache_dl2_lat) do              ]
      dl2_miss++                           ]
  end if                                   ]
  l2_access++                              ]
end if
L1miss = dl1_miss/dl1_access               ]  At the end of a period
L2miss = dl2_miss/dl2_access               ]
```

Instruction-level parallelism (ILP) expresses the average number of operations that can be simultaneously executed in a system. In Algorithm 4.6, we initialized ILPCount parameter to collect the number of operands in each period. In addition to this parameter, simulator holds the number of total ILP calculations with the parameter of TotalILPCalculations. This variable is accumulated during the process of simulator. To gather statistic of the number of ILP, we divide ILPCount to TotalILPCalculations. By doing this calculation, DTABLE is checked to determine the number of operands.

Algorithm 4.6. Pseudo code of the calculation of Instruction-level parallelism (ILP)

```
1:  initialize-parameters                          ]
2:  counter_t ILPCount = 0                         ]  At
3:  counter_t TotalILPCalculations = 0             ]  initialize
4:  TotalILPCalculations++                         ]
5:  if (DTABLEo[i] == DTABLEi1[j]) do              ]
6:    ILPCount++                                   ]
7:  end if                                         ]  Every cycle
8:  if (DTABLEo[i] == DTABLEi2[j]) do              ]  at decode
9:    ILPCount++                                   ]
10: end if                                         ]
11: ILP = ILPCount/TotalILPCalculations;           ]  At  the  end
12: ILPCount = 0                                   ]  of a perdiod
13: TotalILPCalculations = 0                       ]
```

We initialize parameters at first line of Algorithm 4.7. Then, total number of Fetch Queue (FQtot), which holds instructions in fetch phase, is set to 0. To calculate Fetch Queue Occupancy (FQocc), which is the ratio of fullness of fetch with instructions, it is divided by epoch size. In order to accumulate FQtot parameter, it is set to 0 in each iteration. By accumulating FQtot, context[0].FQ_num structure, which belongs to zeroth thread, is implemented because here, we focus on single thread processors.

Algorithm 4.7. Pseudo code of the calculation of fetch queue

```
1: FQtot = 0
2: for each cycle do
3:   FQtot is increased by one
4:   if end of epoch is reached then
5:      FQocc =  FQtot / epoch size
6:      Save FQocc to statistics
7:      FQtot = 0;
8:   end if
9: end for
```

An instruction is decoded or executed in the execution unit. For next instruction, six bytes are kept by instruction queue when it is busy to decode or execute an instruction. These bytes are stored in a first in first out register set. Thus, IQ occupancy directly effects speed of processor by increasing overall efficiency and reduces waiting time for the memory access operations.

We initialize parameters at first line of Algorithms 4.8 through 4.11. Here, IQtot is set to 0 at the beginning of the Algorithm 4.8, and it is divided by epoch size in order to obtain average IQ occupancy (i.e. IQocc). By doing this, contexts[0].icount is implemented because we work with a single thread processor.

Algorithm 4.8. Pseudo code of the calculation of instruction queue occupancy

```
1: IQtot = 0
2: for each cycle do
3:   IQtot+ = contexts[0].icount
4:   if end of epoch is reached then
5:      IQocc = IQtot / epoch size
6:      Save IQocc to statistics
7:      IQtot = 0;
8:   end if
9: end for
```

Re-order buffer controls whether instructions are committed or not. If an instruction is successfully predicted, then it is committed. Otherwise, ROB is flushed. Thus, ROB occupancy is strictly effects on processor performance.

In Algorithm 4.9, the total number of Re-Order Buffer (ROBtot), which holds all in-flight instructions in program order, is set to 0 at the beginning of the code. To calculate Re-Order Buffer occupancy (ROBocc), which is the ratio of buffer utilization by instructions, it is divided by epoch size. In order to accumulate ROBtot parameter, it is set to 0 in each iteration. By accumulating ROBtot parameter, context[0].ROB_num structure, which belongs to zeroth thread, is implemented because here, we focus on single thread processors.

Algorithm 4.9. Pseudo code of the calculation of re-order buffer occupancy

```
ROBtot = 0
for each cycle do
  ROBtot+ = contexts[0].ROB_num
  if end of epoch is reached then
     ROBocc =  ROBtot / epoch size
     Save ROBocc to statistics
     ROBtot = 0;
  end if
end for
```

In Algorithm 4.10, the total number of Load/Store Queue (LSQtot), which holds memory instructions, is set to 0. To calculate Load/Store Queue Occupancy (LSQocc) which is the ratio of fullness of load/store queue, it is divided by epoch size. In order to accumulate LSQtot parameter, it is set to 0 in each iteration. By accumulating LSQtot, context[0].LSQ_num structure, which belongs to zeroth thread, is implemented because here, we focus on single thread processors.

Algorithm 4.10. Pseudo code of the calculation of load/store queue occupancy

```
LSQtot = 0
for each cycle do
  LSQtot+ = contexts[0].LSQ_num
  if end of epoch is reached then
     LSQocc =  LSQtot / epoch size
     Save LSQocc to statistics
     LSQtot = 0;
  end if
end for
```

An arithmetic logic unit is the part of the processor so that it effects processor performance by carrying out arithmetic and logic operations on the operands. In Algorithm 4.11, the total number of Arithmetic Logic Unit (ALUtot), which holds arithmetic and logic operations, is set to 0. To calculate Arithmetic Logic Unit Occupancy (ALUocc), which is the ratio of fullness of operands, it is divided by epoch size. In order to accumulate ALUtot parameter, it is set to 0 in each iteration. By accumulating ALUtot, context[0].ALU_num structure, which belongs to zeroth thread, is implemented because here, we focus on single thread processors.

Algorithm 4.11. Pseudo code of the calculation of arithmetic logic unit occupancy

```
ALUtot = 0
for each cycle do
  ALUtot is increased by one
  if end of epoch is reached then
     ALUocc =  ALUtot / epoch size
     Save ALUocc to statistics
     ALUtot = 0;
  end if
end for
```

A CPU using branch prediction only executes statements if a predicate is true and thus, branch predictors play a critical role in achieving high effective processor performance. In our study, we also integrate the branch misprediction rate as one of the features training our model.

Instructions are executed when they are ready in an instruction window. Here, the number of ready instructions is calculated by checking ready queue, which is a queue of all instructions waiting to be scheduled on a processor. In Algorithm 4.12, we show how we account the average number of ready instruction in the ready queue. Thanks to if/else structure, we check whether integer or floating instructions are ready or not at destination register. Then, ready_inst[0] (integer instructions) and ready_inst[1] (floating instructions) are increased by one. In each period, these arrays are divided by epoch size to calculate the number of them and again, ready_inst[0] and ready_inst[1] are set to 0 in order to gather dynamic result of parameters.

Algorithm 4.12. Pseudo code of the calculation of ready instruction

```
initialize-parameters
counter_t ready_inst[2] = {0, 0}
structure RS_link *link
for (link = ready_queue; link!=NULL; link=link->next) do
   if (RSLINK_VALID(link)) then
      struct ROB_entry *rs = RSLINK_RS(link);
      if (rs->dest_format == REG_INT) then
           ready_inst[0]++
      end if
      else if (rs->dest_format == REG_FP) then
           ready_inst[1]++
      end if
   end if
end for
readyi_int = ready_inst[0]/epoch_size
readyi_fp = ready_inst[1]/epoch_size
ready_inst[0] = 0
ready_inst[1] = 0
```

In general, register allocation is the process of stating a large number of program variables onto a small number of processor registers. Processor runs faster and has better performance when more variables can be in CPU. Instead of memory, processor uses registers so that it fetches faster. However, registers are limited in many processors. Therefore, compiler must decide how to allocate variables. Here, calculation of how many instructions are allocated in register file is crucial to observe the effect of it on processor performance.

We implement Algorithm 4.13 below that PRF_allocated and Total_PRF_allocated arrays are set to 0 at the beginning of the algorithm. Then, allocated instructions are determined by checking dest_format and pyhs_reg. After arithmetic calculations, Total_PRF_allocated array is divide by epoch size so that we collect accumulated allocated integer and floating instructions.

Algorithm 4.13. Pseudo code of the calculation of register file allocation

```
 1: initialize-parameters
 2: counter_t PRF_allocated[2] = {0, 0}
 3: counter_t Total_PRF_allocated[2] = {0, 0}
 4: if (rs->dest_format == REG_INT) then
 5:   PRF_allocated[0] --
 6: else if (rs_dest_format == REG_FP) then
 7:   PRF_allocated[1] --
 8: end if
 9: if (ex_phys_reg == REG_INT) then
10:   PRF_allocated[0] --
11: else if (ex_phys_reg == REG_FP) then
12:   PRF_allocated[1] --
13: end if
14: allc_int = Total_PRF_allocated[0]/epoch_size
15: allc_fp = Total_PRF_allocated[1]/epoch_size
16: Total_PRF_allocated[0] = 0
17: Total_PRF_allocated[1] = 0
18: Total_PRF_allocated[0]+=PRF_allocated[0]
19: Total_PRF_allocated[1]+=PRF_allocated[1]
```

There are two differences to calculate valid integers and floating points in Algorithm 4.14 when it compares to register file allocation. Firstly, at the 8th and 11th line of the algorithm, instructions, which go through the writeback stage, are controlled so that we determine the valid ones. Secondly, if they are within writeback stage, PRF_valid array is increased by one. Therefore, we collect statistics about register file validation.

Algorithm 4.14. Pseudo code of the calculation of register file validation

```
initialize-parameters
counter_t PRF_valid[2] = {0, 0}
vld_int = PRF_valid[0]/epoch_size
vld_fp = PRF_valid[1]/epoch_size
PRF_valid[0] = 0
PRF_valid[1] = 0
**for** (i = 0; i < rf_size; i++) **do**
  **if** (int_reg_file[i].state == REG_WB) **then**
      PRF_valid[0]++
  **end if**
  **if** (fp_reg_file[i].state == REG_WB) **then**
      PRF_valid[1]++
  **end if**
**end for**
```

5. EXPERIMENTAL FRAMEWORK

We use the M-Sim simulator to run SPEC CPU2006 benchmarks on a 4-way out-of-order superscalar processor [9]. Data is collected in every one million cycle. As suggested in the literature, we divide our data set into two portions [11]. First 80 per cent of the data is selected as the training set and the remaining 20 per cent of the data is used for testing the function, which we obtain by running the algorithm described in the previous section. As a result, we run benchmarks at different program regions by fast-forwarding simulations from 80 million to 2 billion cycles, and collect around 2500 training and 800 testing data.

The linear, quadratic, and cubic regression models are applied to first three and first four features, separately in Table 5.2. The algorithms of mathematical functions are implemented in C language using Octave environment. The details of the simulated processor are given in Table 5.1.

Table 5.1. Configuration of the simulated processor

Parameter	**Configuration**
Machine Width	4-wide fetch/dispatch/issue/commit
L/S Queue size	48 Load/Store queue
ROB & IQ size	128 entry ROB, 32-entry IQ
L1 I-cache	64KB, 2-way set-associative 64-byte line
L1 D-cache	64KB, 4-way set-associative 64-byte line, write-back, 1-cycle access latency
L2 Cache unified	512KB, 16-way set-associative 64-byte line, write-back, 10-cycle access latency
BTB	512 entry, 4-way set-associative
Branch Predictor	Bimod: 2K entry
Memory	32-bit wide, 300 cycles access latency

Fig.5.4 and Fig.5.5 show the collected statistics for L1 and L2 caches, respectively. From this figure, we can vaguely identify the relation between the DIPC and L1/L2 cache miss rates. Fig.5.4 also shows that with only one or two features, the machine learning algorithm would not be that successful, since there is a huge variation in DIPC for the same level of cache miss rate. These two figures prove that we need more distinct features to train our model to be more successful in our performance predictions.

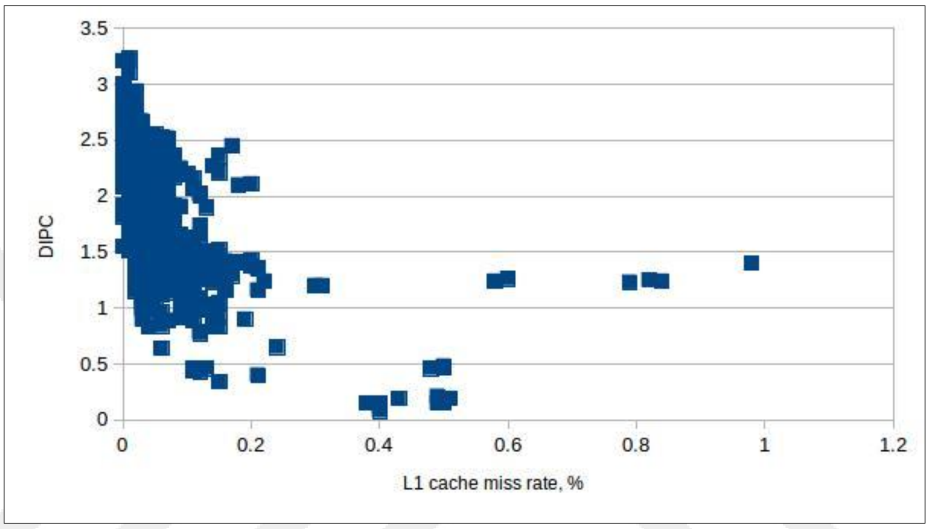

Figure 5.4. Collected statistics for the L1 cache

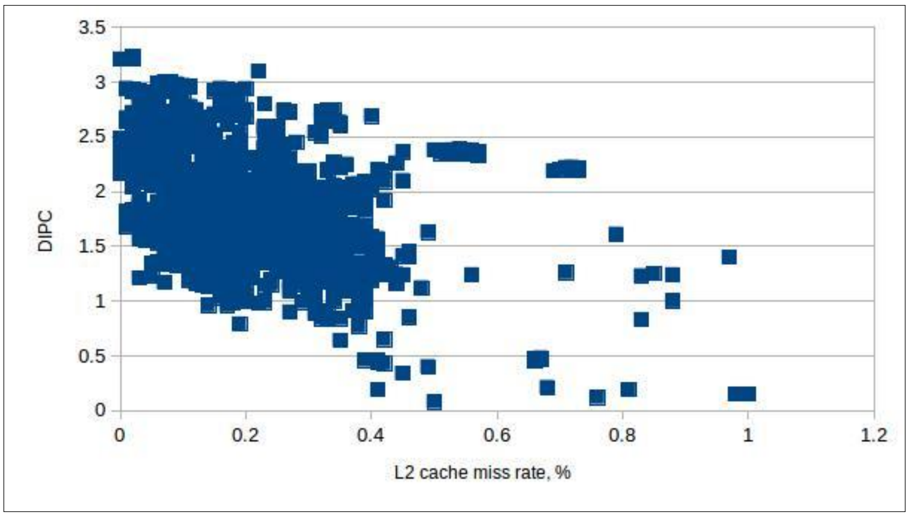

Figure 5.5. Collected statistics for the L2 cache

As a result, we collect 18 processor features that are related to the behavior of running applications. Table 5.2 lists all these features that are used in our machine learning models. Here, ILP is measured as the number of instructions that can be run in parallel in a single cycle. It is collected from the dependency checking logic, which is run at the processor decode stage. Features 6 through 10 are average occupancy levels of Fetch Queue (FQ), Issue Queue (IQ), Re-Order Buffer (ROB), Load/Store Queue (LSQ) and Arithmetic Logic Units (ALU). Then, we collect the number of ready instruction in IQ for both integer and floating-point instructions, separately. Features 13 through 16 are collected from Physical Register File (PRF). First, we collect the average number of allocated PRF entries for integer and floating-point instructions. Then, we also collect the average number of valid PRF entries, as well. Final, two features show the success of speculation done in hardware. Basically, the average number of squashed instructions from the ROB structure shows the efficiency of the processor. When these two values are high, we see a huge drop in IPC, since the speculation mechanism starts throwing all mispredicted instructions into thrash rather than completing them.

Table 5.2. List of features

No	**Feature**
1	L1 miss rate
2	L2 miss rate
3	Dynamic instruction per cycle (DIPC)
4	Instruction per cycle (IPC)
5	Instruction Level Parallelism (ILP)
6	Average Fetch Queue (FQ) occupancy
7	Average Issue Queue (IQ) occupancy
8	Average Re-Order Buffer (ROB) occupancy
9	Average Load/Store Queue (LSQ) occupancy
10	Average Arithmetic Logic Unit (ALU) occupancy
11	Average number of ready instructions (integer)
12	Average number of ready instructions (float)
13	Average number of allocated PRF (integer)
14	Average number of allocated PRF (float)

15	Average number of valid PRF (integer)
16	Average number of valid PRF (float)
17	Average number of ROB squashed (integer)
18	Average number of ROB squashed (float)

6. TEST AND RESULTS

In this section, we present the results for both the regression and the neural network models that we proposed in this study.

6.1. RESULT OF THE REGRESSION MODEL

We used linear, quadratic, and cubic regression models to accurately estimate the DIPC. Based on these methods, we measured four different prediction schemes: (i) DIPC trend (i.e. is performance increasing or decreasing?), (ii) 3 per cent DIPC range (i.e. performance prediction is assumed to be correct if the predicted DIPC is still within 3 per cent range of the actual DIPC), (iii) 5 per cent DIPC range, and (iv) 10 per cent DIPC range (same prediction scheme for 5 per cent and 10 per cent range, respectively). In Fig.6.6, we show our prediction results for linear, quadratic, and cubic regression models. Here, the configurations for the predictions are represented by X/Y notation, where X represents the prediction scheme and Y represent the number of features. As we see from the figure, increasing the number of features has a positive effect on prediction accuracy (number of correct guesses/number of total guesses). Secondly, as we expected, the width of the DIPC range has an important role on the prediction accuracy, as well. As we relax the range, we observe higher prediction accuracy.

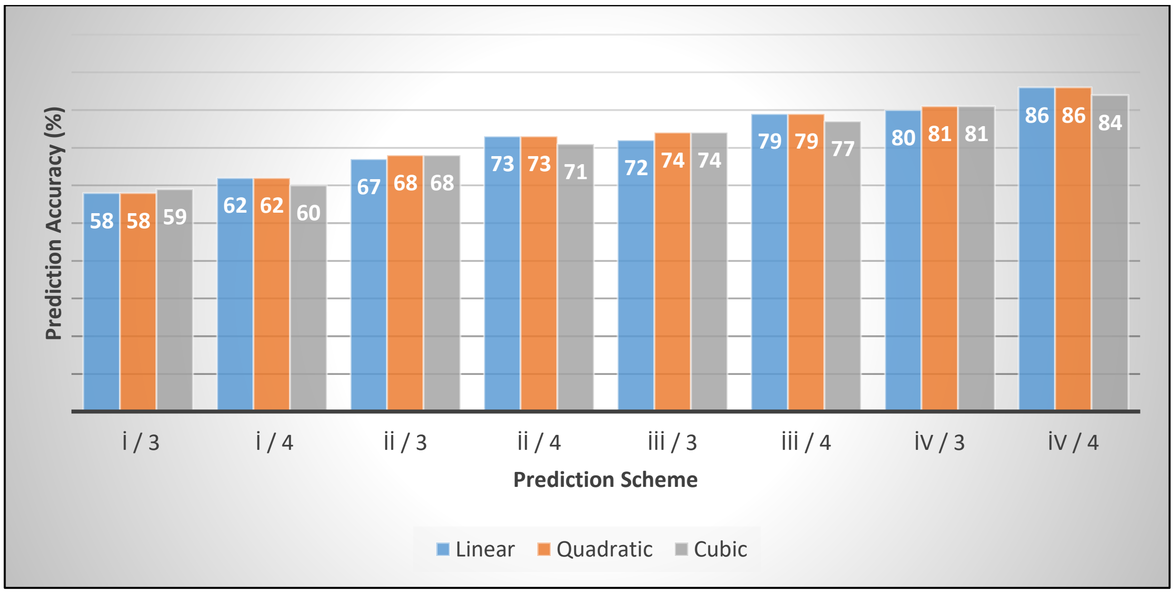

Figure 6.6. Prediction accuracy for the linear regression model

An important observation might be the insensitivity of the accuracy results to different regression models. We see that quadratic regression model performs slightly better than the other models. However, even the simplest linear model works fine. We do not see a radical change when we change our regression model from linear to polynomial.

Here, we closely focus on graphs of three different models on prediction accuracy; normal equation, gradient descent and normalization. According to prediction results above Fig.6.6, we obtain quadratic regression model is very slightly better than linear and cubic regression models. We acquire utmost prediction results for quadratic regression model by applying normal equation in Fig.6.7 and Fig.6.8, as well. Fig.6.7 is the graph, which is obtained by testing on first 20 per cent part of the whole data.

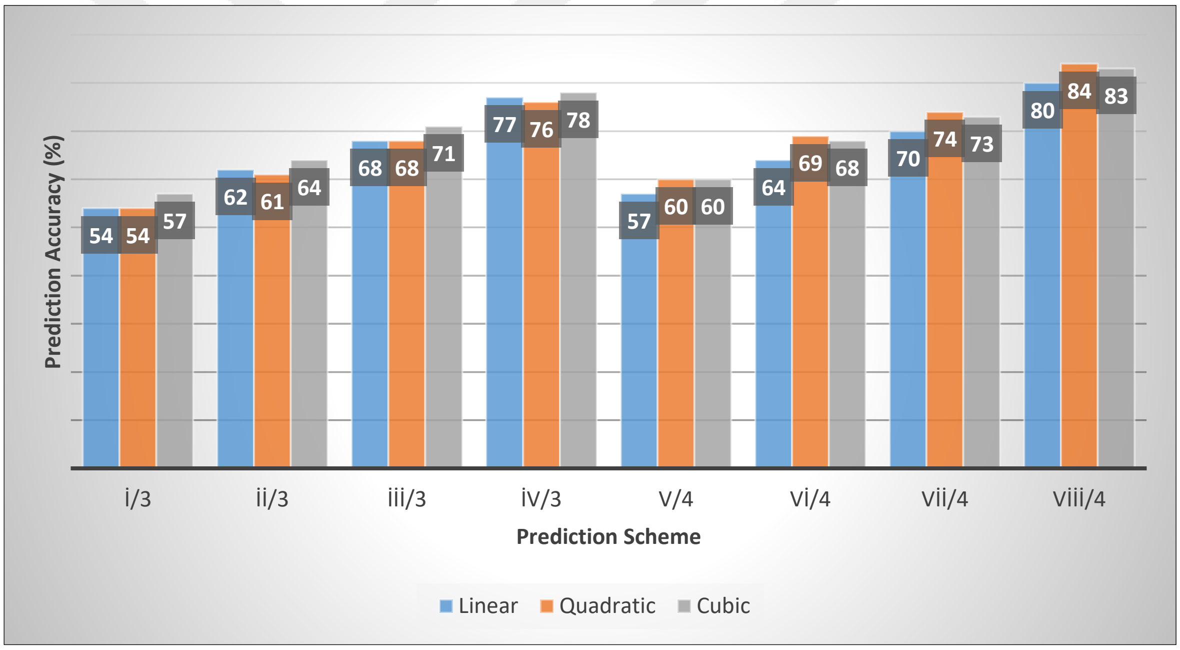

Figure 6.7. Prediction accuracy of normal equation – first testing data

On the other hand, in Fig.6.8, prediction results are gathered by testing on the last 20 per cent part of the whole data. As you see from graphs, the number of features has significantly effects on processor performance. Secondly, prediction accuracy rises with an increasing in width. Common point of two graphs is to reach top accuracy score for 4 features quadratic regression model.

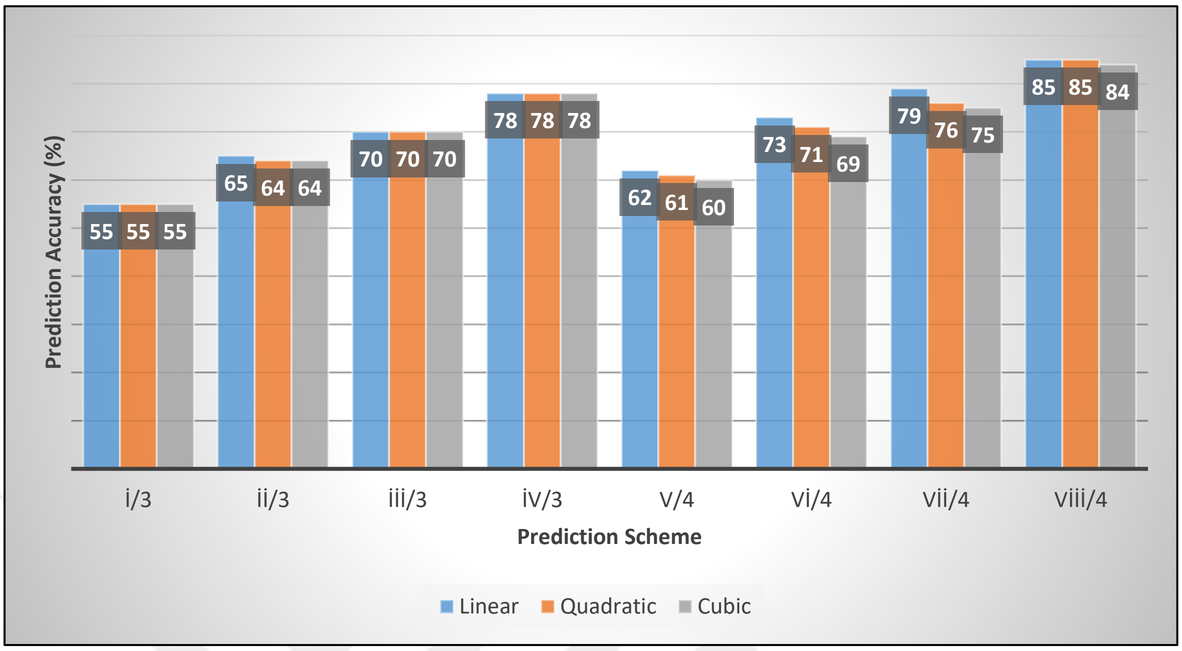

Figure 6.8. Prediction accuracy of normal equation – second testing data

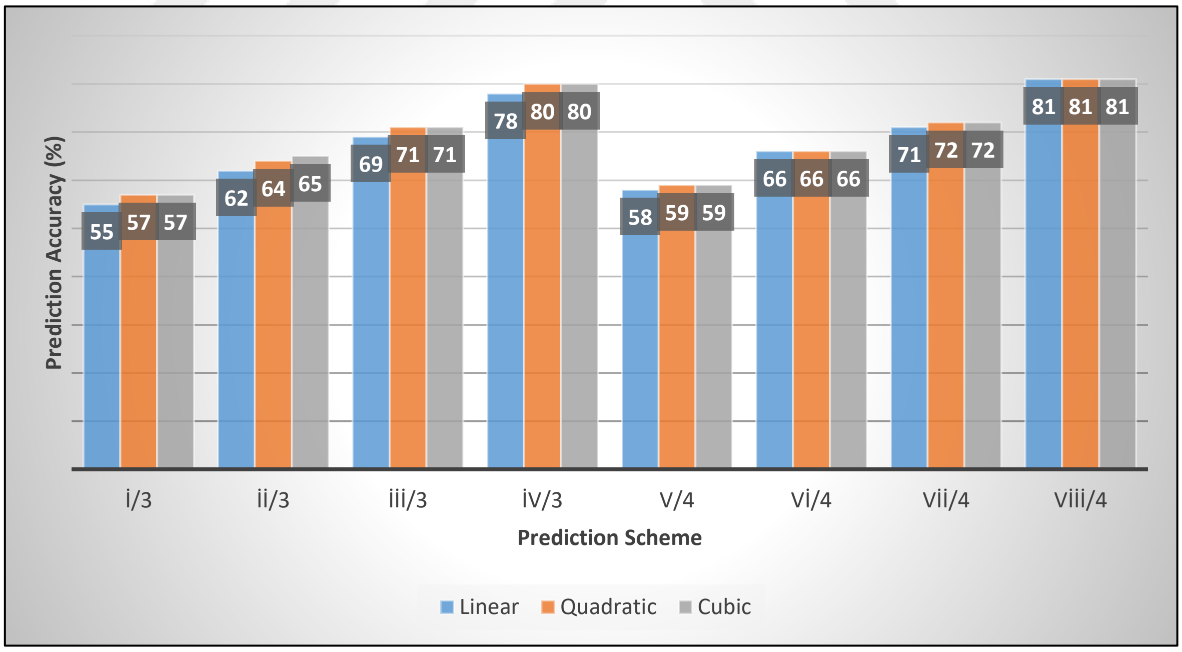

Figure 6.9. Prediction accuracy of gradient descent – first testing data

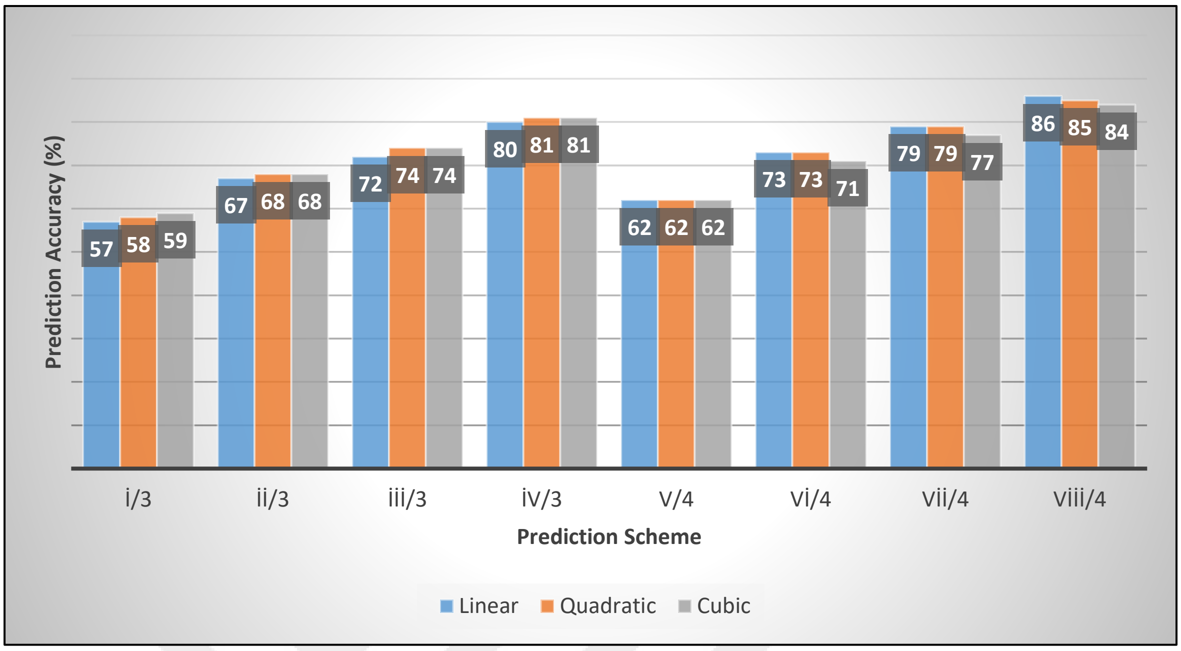

Figure 6.10. Prediction accuracy of gradient descent – second testing data

In Fig.6.9 and Fig.6.10, graphs of prediction results are formed by applying gradient descent model to NN system. It is clear that in both graphs, we obtain high processor performance for 4 features and 0.1 width range though percentage of prediction accuracy is almost equal for 3 features and 0.1 width range. As you see from graphs, we get better accurate results with the last 20 per cent of testing data in Fig.6.10 by using gradient descent model.

There is a considerable diversity in Fig.6.11 and Fig.6.12, which are obtained by normalization method. Increasing the number of features from 3 to 4 has effects on prediction accuracy in as positive way. In contrast, here, prediction accuracy slightly decreases when the number of features is raised to 4. Secondly, it is reached to top point with cubic regression model unexpectedly and there is no crucial difference between figures apart from the percentage of quadratic regression model. The prediction accuracy of quadratic regression model stays under expectation in Fig.6.11 though it is almost close to linear and cubic regression models in Fig.6.12. Normalization algorithm cannot adequately capture the underlying structure of our last feature data so that under-fitting occurs in Fig.6.11. Normalization is the method that we receive low prediction accuracy results in almost each model when we compare to normal equation and gradient descent. However, it is important to observe how prediction accuracy changes while the number of features increase and the whole data normalize.

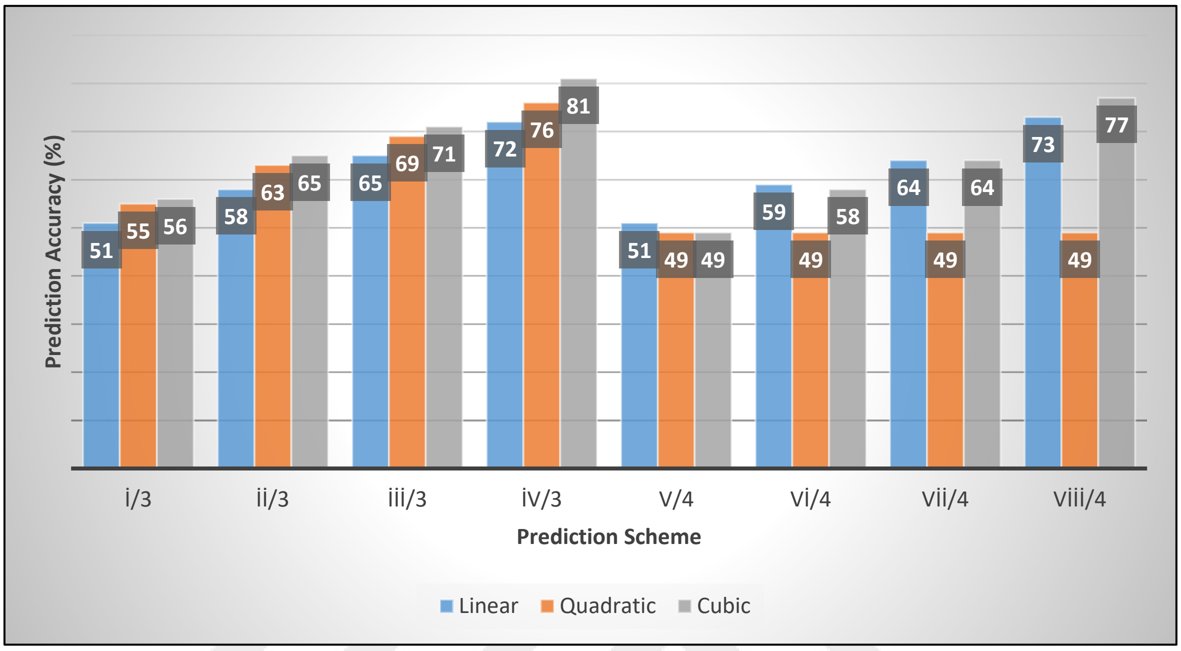

Figure 6.11. Prediction accuracy of normalization – first testing data

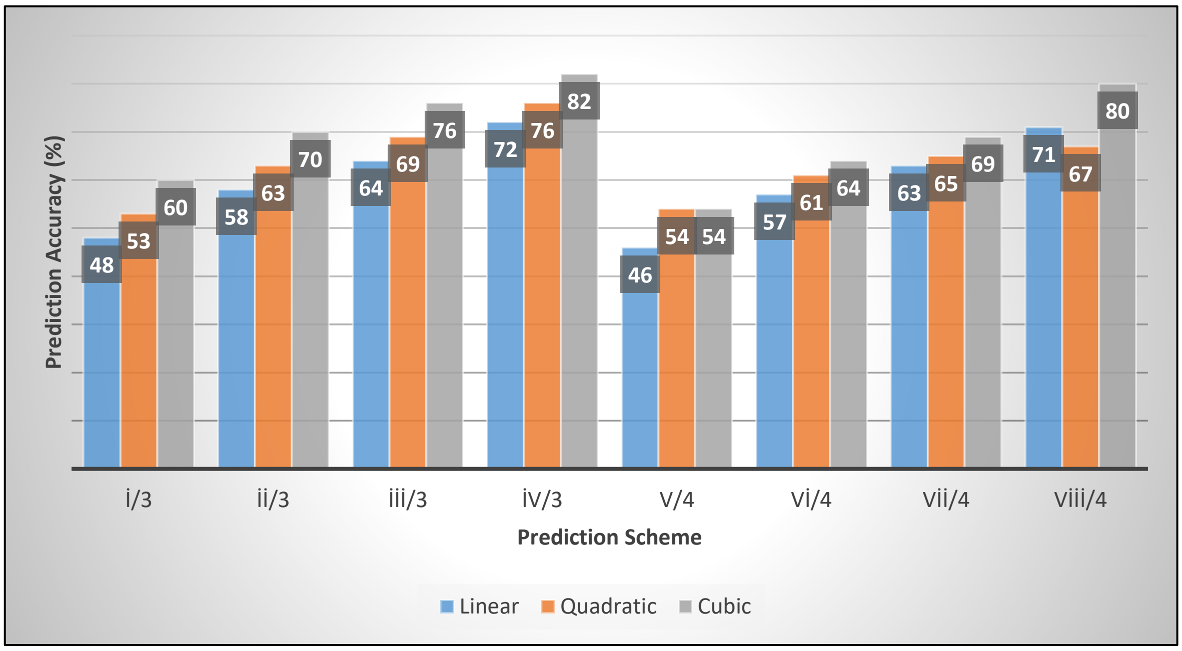

Figure 6.12. Prediction accuracy of normalization – second testing data

6.2. RESULT OF THE NEURAL NETWORK MODEL

NN graphs, which are obtained by iterating or processing the NN system, from 100 to 1500 iterations show prediction results for 4, 10, and 18 features. Each iteration contains 16 prediction accuracy results with different regularization parameters, lambda values and various number of hidden nodes. Results with lambda value of 0 are represented by yellow, with lambda value of 0.1 are represented by green, with lambda value 0.5 are represented by dark blue, and, finally, with lambda value of 1.0 are represented by magenta colored nodes. As shown in Fig.6.13, when we consider 100 iterations experiment, it is divided into four regions including aforementioned lambda values for four various hidden layer nodes (1, 2, 4 and 6). For the sake of clarity, we only show these regions for 100 iteration results on the graph. The same presentation method is also applied to Figures 6.13 through 6.18.

When we consider Fig.6.13 and 6.14, it is clear that normalization method has a positive effect on processor performance for 4 features. In Fig.6.13, we acquire the widest range of prediction accuracy results and reach to a peak accuracy at 67 per cent with 300 iterations, 6 hidden nodes and lambda value of 0 configuration. At all points in Fig.6.13, percentage of prediction accuracy is better than without normalization method.

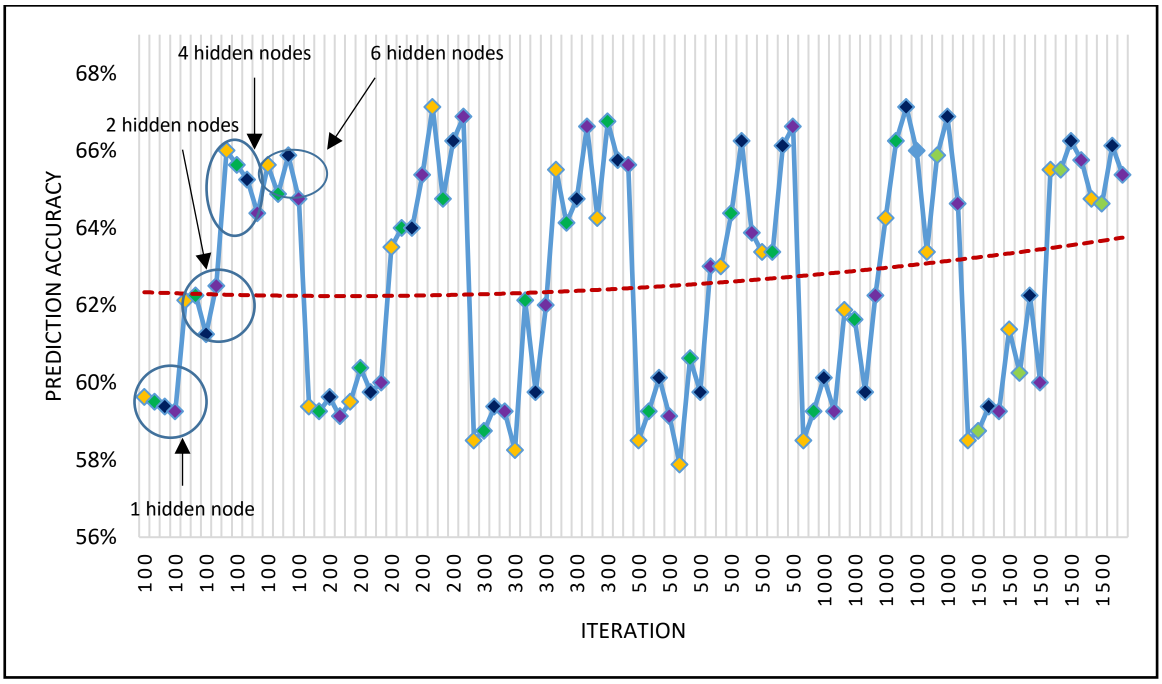

Figure 6.13. Prediction accuracy for 4 features – with normalization

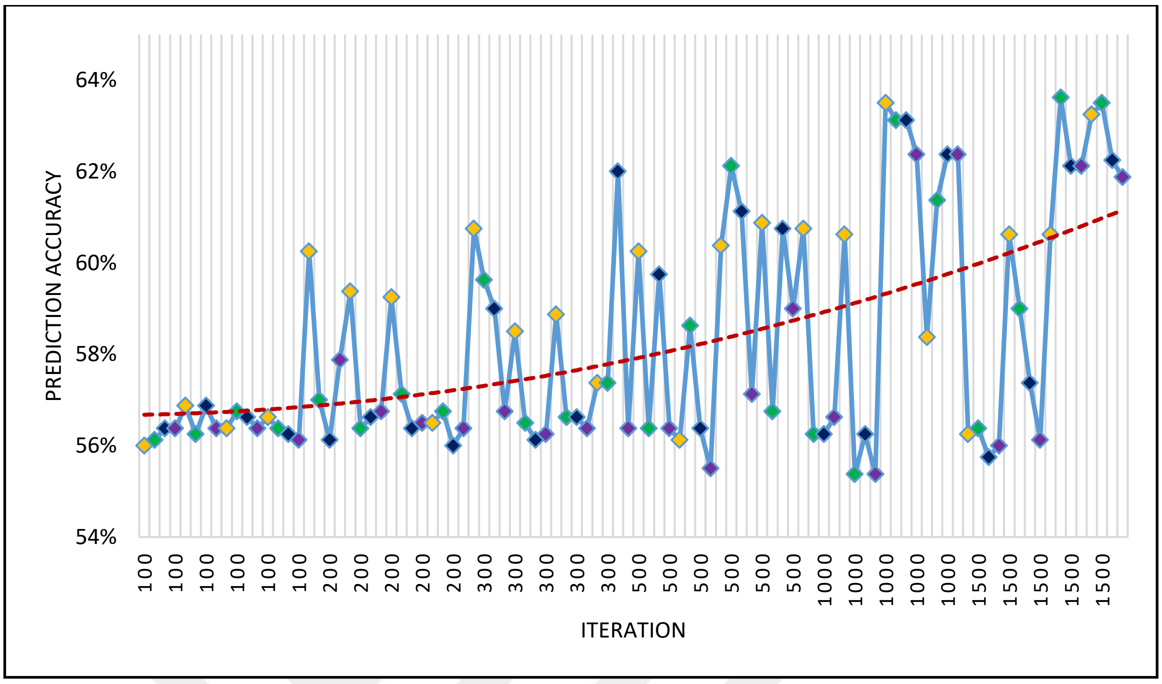

Figure 6.14. Prediction accuracy for 4 features – without normalization

In Fig.6.15 and Fig.6.16, normalized and non-normalized NN systems work fine for 10 features for 5, 9, 10 and 12 hidden layer nodes. However, there are some obvious differences when we compare these two graphs.

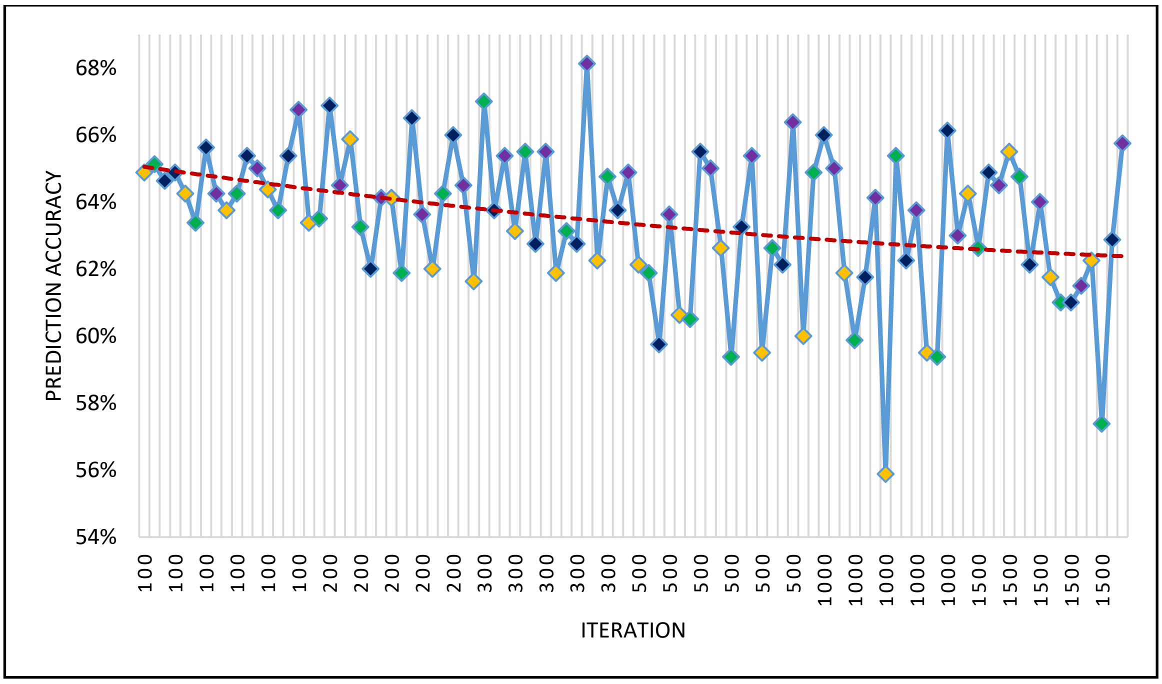

Figure 6.15. Prediction accuracy for 10 features – with normalization

Firstly, we observe that the NN system outputs advanced prediction results at 300 iterations. It is sufficient to use 10 hidden units in order to reach to 68 per cent of accuracy with normalization method. Secondly, fluctuation of each iteration in Fig.6.16 stays growing positon in contrast to Fig.6.15 that has a decrease trend after 300 iterations. NN system reaches to pick point within 1500 iterations for without normalization as 66 per cent.

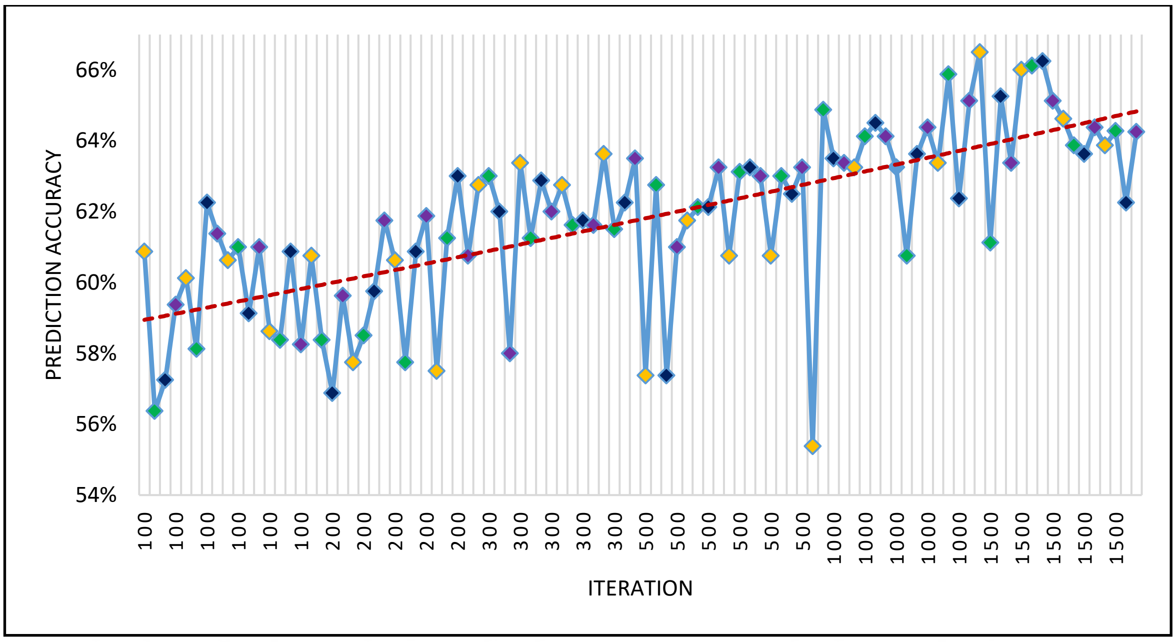

Figure 6.16. Prediction accuracy for 10 features – without normalization

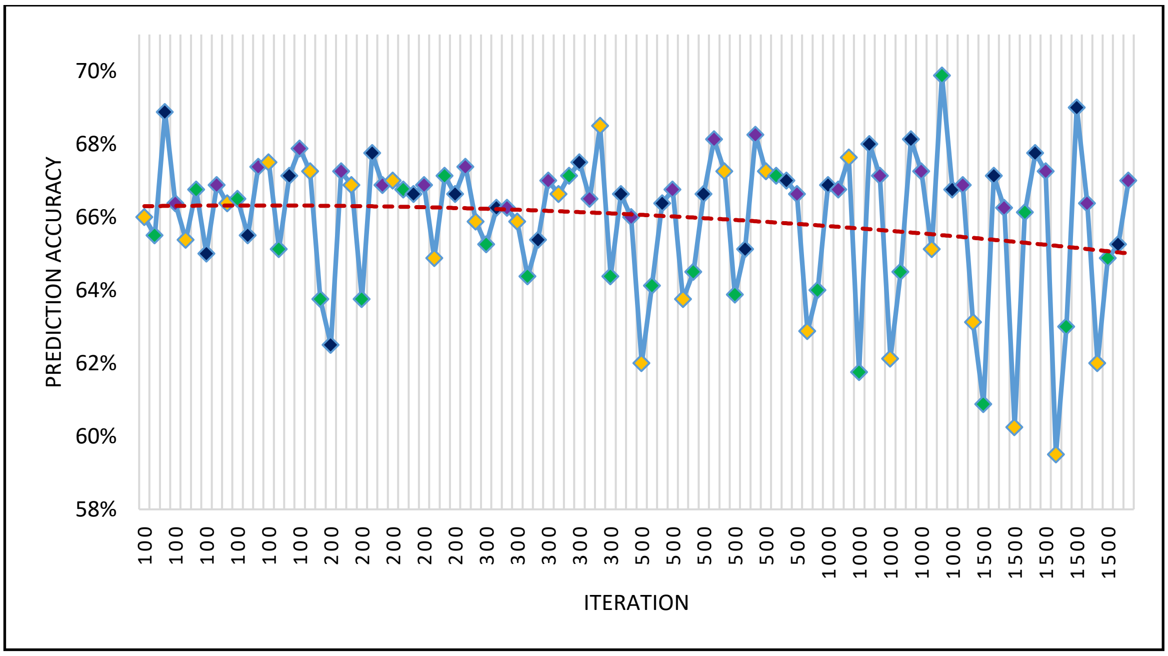

Figure 6.17. Prediction accuracy for 18 features – with normalization

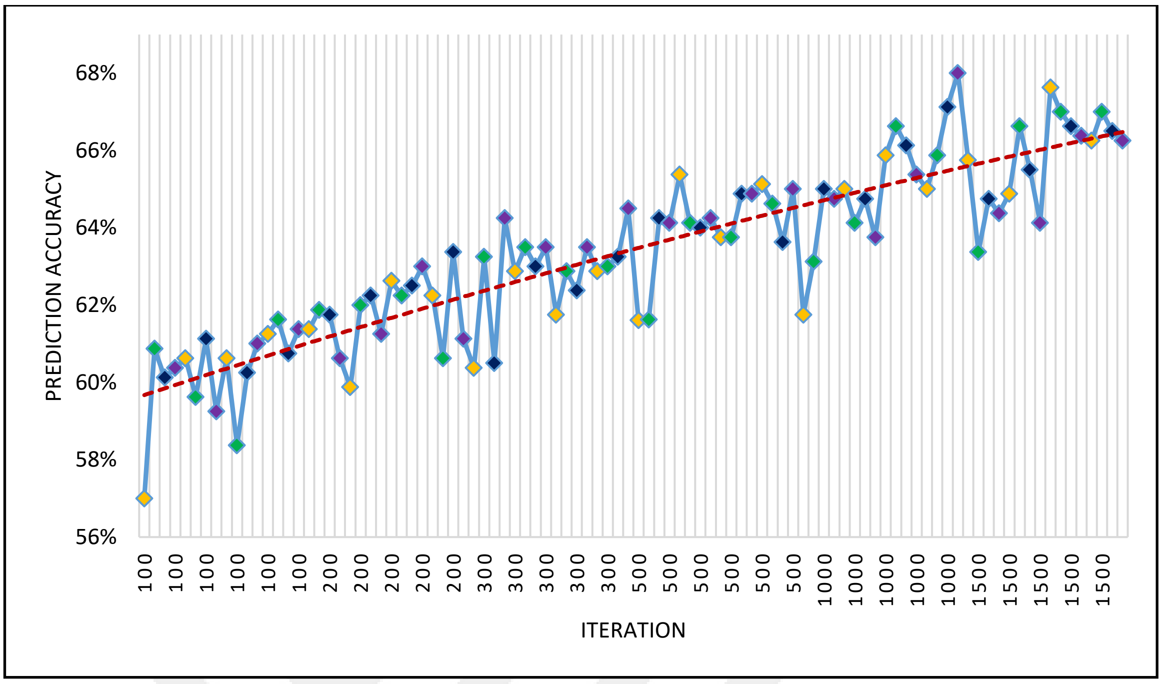

Figure 6.18. Prediction accuracy for 18 features – without normalization

In Fig.6.17 and 6.18, 9, 13, 18, 20 hidden layer nodes are used. As we expected, we obtain utmost result in normalized NN system by testing 18 features as 70 per cent in Fig.6.17. We reach to that result at 1000 iterations. It is obvious that there is a significant percentage differences between two graphs. For 100 iterations, prediction accuracy of normalized graph is almost 64 per cent although it is not even close to 59 per cent in Fig.6.18.

When we consider al graphs above, increasing the prediction accuracy is systematical that it scales from 64 per cent to 67 per cent for without normalization and that rising trend maintains its position for normalization, as well (from 67 per cent to 70 per cent). We see precise and accurate changes when we modify the number of features and hidden units. Thus, it is certain that processor performance is directly proportional to the increase on features and hidden units.

6.3. COMPARISON BETWEEN REGRESSION AND NN MODELS

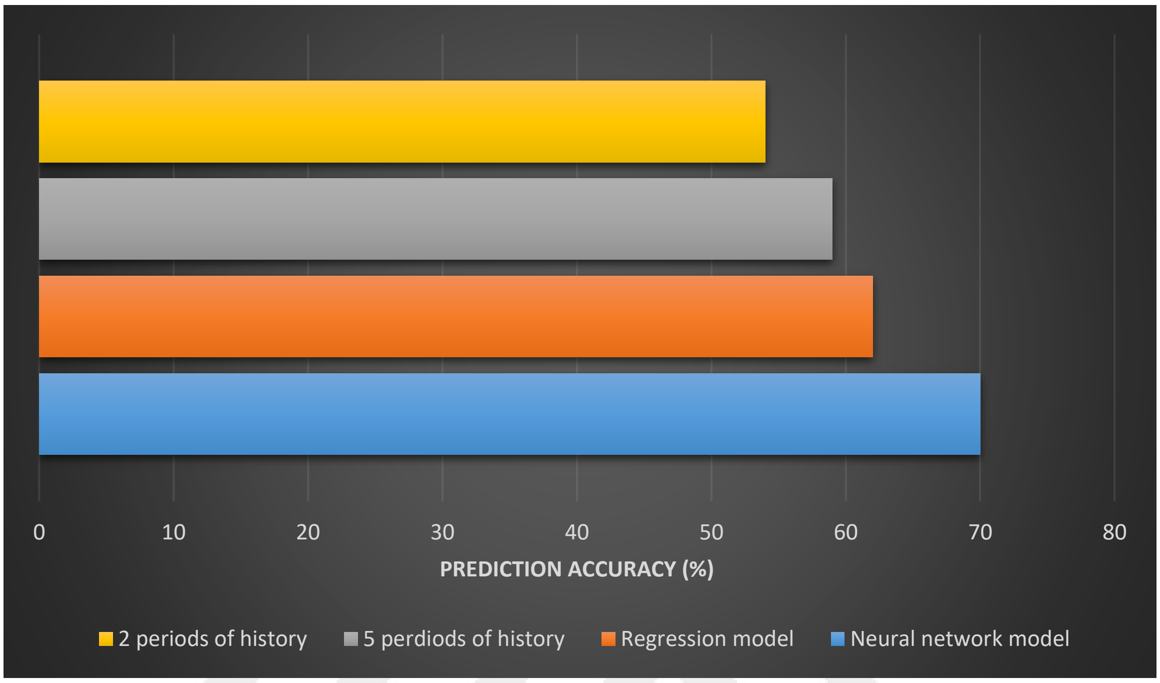

Figure 6.19. Comparison between regression and NN models

Fig.6.19 describes the comparison between regression model and neural network in terms of prediction accuracy. The highest estimated result between linear, quadratic, and cubic regression models are illustrated in this figure and its result compared with normalized neural network. As you can see from the figure, prediction accuracy shows a significant rise from 62 per cent to 70 per cent. We find normalized neural network with twenty features the most promising model in this study.

In Fig.6.19, we predicted the performance accuracies of the baseline algorithms with two or five periods of history. We obtained 54 per cent and 59 per cent prediction accuracies by considering trends (increasing or decreasing) of two or five periods of data history. Whereas results are better than random guess (50 per cent), they are less than regression and NN models, as we expected.

We trained our NN model with a various set of features (4, 10, and 18). First four features, which are given in Table 5.2, make the first set of features. Note that these four features are also used to train our regression model. Next, we used first ten and all eighteen features to train our NN model. Fig.6.20 shows the accuracy results for all these settings with and without normalization of features. As we see from the figure, when the number of features is increased, the prediction accuracy of the NN gets better and better. We also find that normalization always helps improving the accuracy level of the NN predictor, since there are a variety of features with different minimum and maximum ranges. We see that the maximum prediction accuracy of 70 per cent is observed when the number of features reach to eighteen with the normalization is in effect.

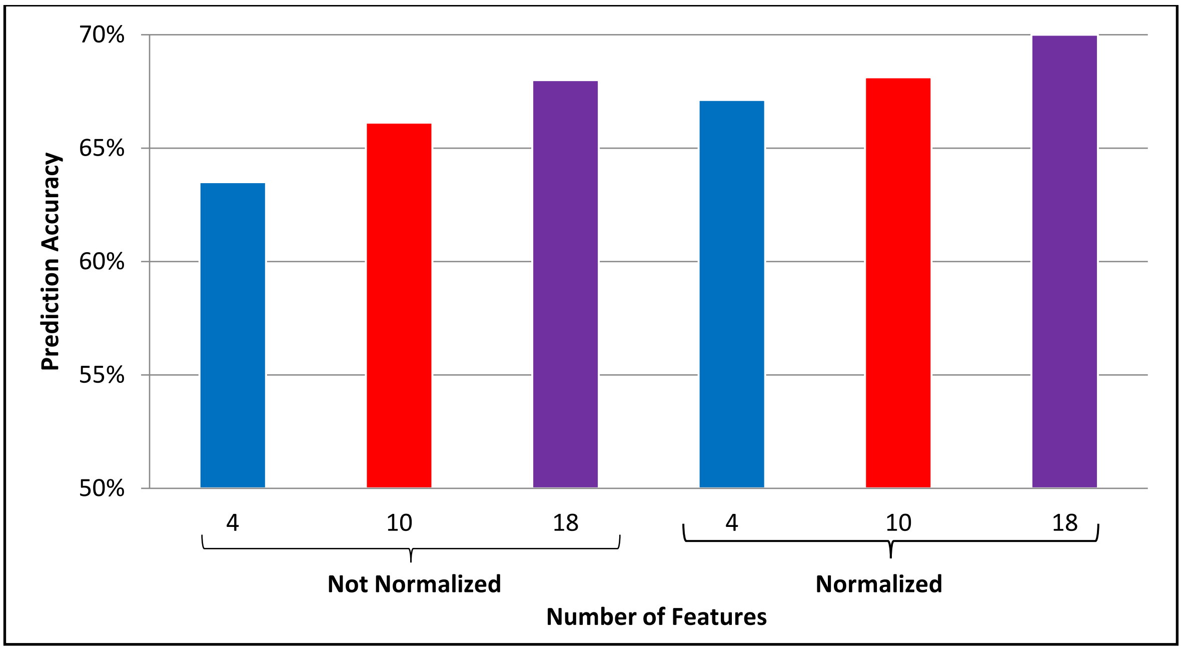

Figure 6.20. Prediction accuracy for the NN model with and without normalization applied on features

We try to explore the search space by varying the number of hidden layers, the number of iterations, the value of lambda, and the number of features and also by applying normalization to the training data. The following Fig.6.21 and Fig.6.22 show the isolated effect of each of these parameters on the prediction accuracy. For example, Fig.6.21 shows the effect of the number of hidden layer nodes on the prediction accuracy when the other parameters are kept constant. From this figure, we see that the prediction accuracy can be improved by increasing the number of hidden layer nodes in a neural network. This conclusion is not true for all the cases, though. We see that the maximum number of hidden layer nodes for 4 and 10 features do not give the best accuracy.

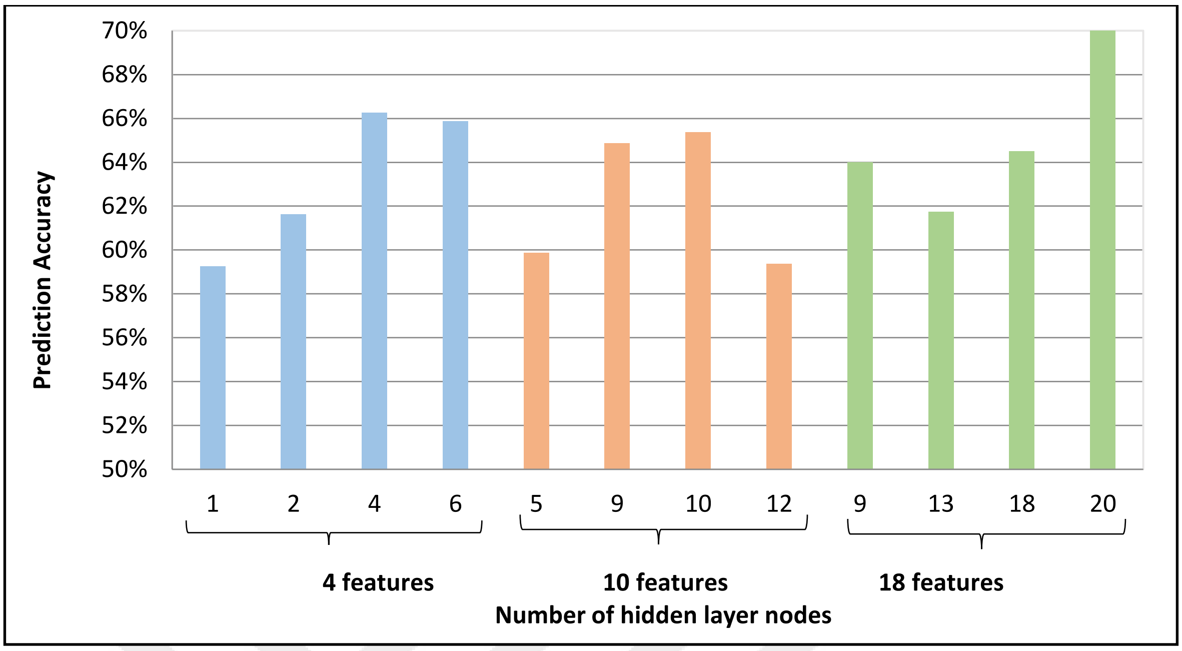

Figure 6.21. Prediction accuracy for various number of hidden layer nodes

Fig.6.22 shows the effect of the regularization parameter, lambda, on the prediction accuracy when the other parameters are kept constant. We only studied four different values of the lambda in our tests, and either 0.1 or 0.5 seems to work fine to achieve the best accuracy with different number of features.

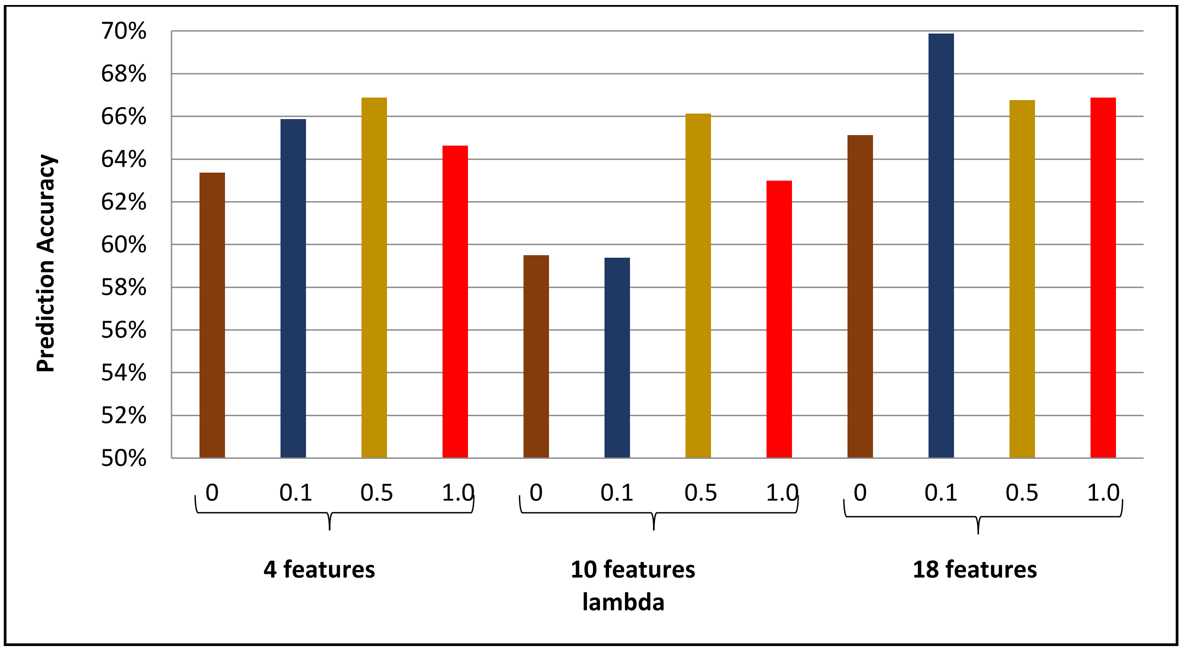

Figure 6.22. Prediction accuracy for various lambda values

Normally, we used two types of data sets to perform supervised neural network. In one dataset which is called training data, we have input data together with expected output. This dataset is prepared by collecting some data in benchmarks. Here, we have expected output for every data row. In second dataset which is called testing data, this is the data in which we are interested for the output of our model and thus the data include unexpected dataset. Finally, we decided to test our training data with obtained weights in order to estimate how well our model has been trained. The accuracy of model depends on the size of our data and the value we would like to predict. Thus, we obtained 73 per cent prediction accuracy on training dataset, which is, it is slightly accurate than ever the prediction of 18 features and 20 hidden layer nodes. At some point, NN stops learning useful general features and iteration number or regularization parameter might cause that issue. In the future, NN can be trained with different regularization coefficients or can be expanded the training and feature dataset. Furthermore, NN model might not be sufficient leariining capacity so that number of hidden layers or number of hidden layer nodes can be altered.

7. CONCLUSION

In this study, we focus on the accurate performance prediction of applications running on out-of-order superscalar processors. We studied a variety of machine learning methods including the normal equation, the gradient descent, the normalization algorithm with linear, quadratic and cubic regression models, and, finally, the neural network model with various number of processor features. The baseline algorithms with two or five periods of history achieve prediction accuracies of 54 per cent and 59 per cent, respectively. We find that the best prediction accuracy with a regression model is achieved with the quadratic regression model at 62 per cent. Finally, the neural network model achieves much better results even with the same number of features, and it seems more suitable to problems similar to the one studied in this thesis. The best prediction accuracy (70 per cent) is achieved with a neural network trained with 18 normalized features.

REFERENCES

1. Jimenez DA, Lin CT. Dynamic branch prediction with perceptrons. *In High-Performance Computer Architecture. The Seventh International Symposium on;* 2001, February: ACM & IEEE.

2. Joseph PJ, Vaswani K, Thazhuthaveetil MJ. Construction and use of linear regression models for processor performance analysis. *In High-Performance Computer Architecture. The Twelfth International Symposium on*; 2006, February: ACM & IEEE.

3. Lo JL, Parekh SS, Eggers SJ, Levy HM, Tullsen DM. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*; 1999.

4. Monreal T, González A, Valero M, González J, Viñals V. Dynamic register renaming through virtual-physical registers. *Journal of Instruction Level Parallelism*; 2000; 2: 4-16.

5. Choi S, Yeung D. Learning-based SMT processor resource distribution via hill-climbing. *In ACM SIGARCH Computer Architecture News; 2006, June*: IEEE Computer Society.

6. Wang H, Koren I, Krishna CM. An adaptive resource partitioning algorithm for SMT processors. *In Proceedings of the 17th international conference on Parallel architectures and compilation techniques*; 2008, October: ACM.

7. Cazorla F J, Ramirez A, Valero M, Fernandez E. Dynamically controlled resource allocation in SMT processors. *In Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*; 2004, December: IEEE Computer Society.

8. Karkhanis TS, Smith JE. A first-order superscalar processor model. *In Computer Architecture. Proceedings. 31stAnnual International Symposium on*; 2004, June: IEEE.

9. Sharkey J, Ponomarev D, Ghose K. A flexible, multithreaded architectural simulation environment. Dept. of CS, SUNY-Binghamton; October, 2005. Technical Report CS-TR-05-DP01.

10. Ipek E, De Supinski BR, Schulz M, McKee SA. An approach to performance prediction for parallel applications. *In European Conference on Parallel Processing;* 2005, August: Springer, Berlin, Heidelb.

11. Reitermanova Z. Data splitting. *In WDS*; 2010.

12. Balaprakash P, Gramacy RB, Wild SM. Active-learning-based surrogate models for empirical performance tuning. In *Cluster Computing (CLUSTER), International Conference on*; September, 2013: IEEE.

13. Cummins C, Petoumenos P, Wang Z, Leather H. End-to-end deep learning of optimization heuristics. In *Parallel Architectures and Compilation Techniques (PACT), 26th International Conference on*; September, 2017: IEEE.

14. Dubach C, Jones TM, O'boyle MF. Exploring and predicting the effects of microarchitectural parameters and compiler optimizations on performance and energy. *ACM Transactions on Embedded Computing Systems (TECS)*; 2012.