CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Learning the code sequence to solve a given problem can be done through various techniques. There are various forms of machine learning. But for this purpose, we require a powerful learning mechanism that can capture any amount of sequences that have been fed into it. The right learning mechanism for this is deep learning, which can model complicated functions through weights and activation functions. To feed the program sequence into the deep network, the program has to be converted into a numerical representation.

1.2 LOGICAL ERRORS IN C PROGRAMMING

Learners of programming, generally choose to begin with C language because of its simplicity and generality. When a beginner programmer tries to solve a given problem using C program, the programmer might do the any one of the following errors - Syntax error, run time error, and logical error.

Syntax error leads to compilation failure. Run time error includes exceptions (like division by zero, array index out of bounds etc.). Logical error leads to infinite loop or incorrect solution to the problem. In case of compilation failure, the compiler points out all the syntax errors made, and the expected changes. For run time error, there is some indication like “core dumped” or “segmentation fault”. But for logical error, the beginners struggle to figure out what the error is, and how to rectify it.

1.3 DEEP LEARNING

Deep learning is part of a broader family of machine learning methods based on learning representations of data. An observation (e.g., an image) can be represented in many ways such as a vector of intensity values per pixel, or in a more abstract way as a set of edges, regions of particular shape, etc. Some representations are better than others at simplifying the learning task (e.g., face recognition or facial expression recognition). One of the promises of deep learning is replacing handcrafted features with efficient algorithms for unsupervised or semi-supervised feature learning and hierarchical feature extraction.

Research in this area attempts to make better representations and create models to learn these representations from large-scale unlabeled data. Some of the representations are inspired by advances in neuroscience and are loosely based on interpretation of information processing and communication patterns in a nervous system, such as neural coding which attempts to define a relationship between various stimuli and associated neuronal responses in the brain.

Various deep learning architectures such as deep neural networks, convolutional deep neural networks, deep belief networks and recurrent neural networks have been applied to fields like computer vision, automatic speech recognition, natural language processing, audio recognition and bioinformatics where they have been shown to produce state-of-the-art results on various tasks.

1.4 RECURRENT NEURAL NETWORK

The idea behind Recurrent Neural Networks is to make use of sequential information. In a traditional neural network we assume that all inputs (and outputs) are independent of each other. But for many tasks that’s a very bad idea. For e.g., To predict the next word in a sentence there is a necessity to know which words came before it. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a memory which captures information about what has been calculated so far. In theory RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps.

1.5 LONG SHORT-TERM MEMORY RECURRENT NEURAL NETWORK

In a traditional, during the gradient back-propagation phase, the gradient can end up being multiplied a large number of times (as many as the number of timesteps) by the weight matrix associated with the connections between the neurons of the recurrent hidden layer. Because of this, the magnitude of weights in the transition matrix can have a strong impact on the learning process.

If the weights in the matrix are small (or, more formally, if the leading eigenvalue of the weight matrix is smaller than 1.0), it can lead to a situation called vanishing gradients where the gradient signal gets so small that learning either becomes very slow or stops working altogether. It can also make more difficult the task of learning long-term dependencies in the data. Conversely, if the weights in the matrix are large (or, again, more formally, if the leading eigenvalue of the weight matrix is larger than 1.0), it can lead to a situation where the gradient signal is so large that it can cause learning to diverge. This is often referred to as exploding gradients.

These issues are the main motivation behind the LSTM model [1] which introduces a new structure called a memory cell. A memory cell is composed of four main elements: an input gate, a neuron with a self-recurrent connection (a connection to itself), a forget gate and an output gate. The self-recurrent connection has a weight of 1.0 and ensures that, barring any outside interference, the state of a memory cell can remain constant from one timestep to another. The gates serve to modulate the interactions between the memory cell itself and its environment. The input gate can allow incoming signal to alter the state of the memory cell or block it. On the other hand, the output gate can allow the state of the memory cell to have an effect on other neurons or prevent it. Finally, the forget gate can modulate the memory cell's self-recurrent connection**,** allowing the cell to remember or forget its previous state, as needed**.**

1.6 DIFFERENT FORMS OF THE PROGRAM STATEMENTS

In order to feed the program into the deep neural network, we need to represent the program statements or program elements in some format suitable for learning the sequence of the program. For this purpose, we have come up with two different representation forms.

1.6.1 Statements as Vectors

Since any neural network has a definite number of nodes in the input layer, and every statement has variable number of tokens, each statement has to be converted to a vector of a fixed dimension *d*. The vectors should be in such a way that, when two statements are semantically similar to each other, their corresponding vectors should lie closer to each other in the *d* dimensional continuous space. For this conversion, the LSTM network suggested by Palangi et. Al. [15] can be used. Their work represents an English sentence as a vector. The same technique can be experimented for C language. The deep network used to convert the statement to the vector is LSTM. Due to the ability of LSTM networks to capture long term memory, the LSTM-RNN accumulates increasingly richer information as it goes through the sentence, and when it reaches the last word, the hidden layer of the network provides a semantic representation of the whole sentence.

The model is found to automatically attenuate the unimportant words and detect the salient keywords in the sentence. The aim is to train a model that can automatically transform a sentence to a vector that encodes the semantic meaning of the sentence.

1.6.2 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches. The AST for the following program sequence is given in figure 1.1.

```
apple = banana * 2 / carrot;
if (date == eggplant)
{
        fennel = grits + 3.14 * hominy;
        juniper = kale;
}
else
        lettuce = mustard - napa;
```

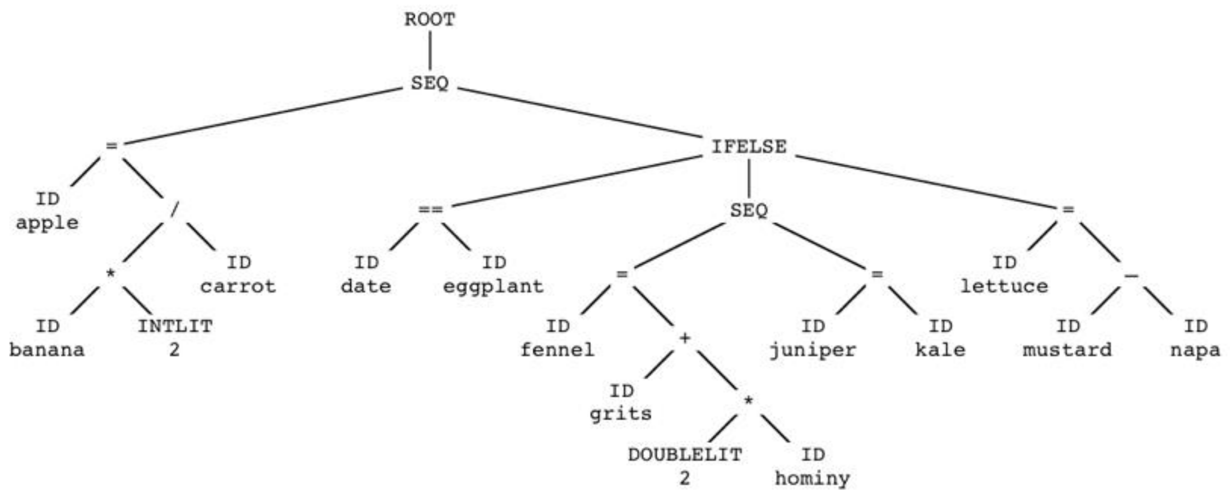

Figure 1.1 Abstract Syntax Tree for Sample Code Snippet

By using the AST of a program, one can get an overall idea of the semantics of the program. Different programs that are solutions to the same problem, using the same algorithm, have similar ASTs. This fact makes the learning of the programs easier. Each node of the tree belongs to one category or class of programming construct in c language. A preorder traversal of the tree, gives a sequence of such node classes, which can be fed into the deep neural network to learn the program.

1.7 COMPARISON OF THE DIFFERENT FORMS

We have seen that the program statements can be represented as a sequence of *d* dimensional vectors or as an Abstract Syntax Tree, in order to feed the program into the network to learn. In the case of using vector representation, the vector representation of statements has to be learnt by another complicated neural network. While any learning always depends only on the data set, and learning does not always comply to give 100% accuracy, it is always better to go for a static method, which is independent of the data provided whenever possible. Hence we leverage the Abstract Syntax Tree representation, which is independent of the syntactic structure, similar for programs with similar goals, does not require training using programs, and thereby does not depend on the programs.

1.8 OBJECTIVE

The goal is to develop an Integrated Development Environment for C programming where a student can choose one problem from the list of problems available, and the teacher is given the option for adding new problems with name, description and test cases. Whenever the student types a code statement that is not likely to give the correct solution for the problem, the IDE should suggest ways of correcting the code such that it gives the correct solution. The IDE is expected to check for the correctness of solution as and when the solutions are submitted for a problem, using the test cases given by the teacher. The correct solutions are added to the training set of the suggestion system for it to learn the new solution.

CHAPTER 2

LITERATURE SURVEY

Different techniques have been proposed so far to solve the existing system. The idea of all these proposals is to apply natural language processing over a programming language. Most of the works targeted and experimented on Java language.

2.1 CODE COMPLETION ENGINES

The existing systems provide to complete the current statement while typing the code. This is widely known as “code completion”. So far, code completion has always meant to complete the current statement, given an incomplete statement, i.e., a sequence of tokens. e.g., if the user types

```
for(i=0; i
```

the code completion engine suggests that the code completes as

```
for(i=0; i<n; i++)
```

There has been good research in the field of code completion. Various techniques have been proposed, that decrease the perplexity produced by the technique used in the previous proposals. Some of the very popular IDEs (Eclipse, Visual Studio, etc.,) have implemented the code completion system successfully. The code completion is also referred to as “IntelliSense”.

2.2 STATISTICAL LANGUAGE MODELING OVER PROGRAMMING LANGUAGE

A lot of language modeling techniques have been applied to programming languages.

The bag-of-words model is a popular language modeling technique, which calculates the probability of every word in the vocabulary to be the next word, given a set of word sequences under a given domain. This is applied to tokens in a programming language instead of words in natural language.

The n-grams method introduces a small variation. It also takes the order of occurrence of the tokens into account as opposed to the bag-of-words, which just decides based on the count of every word. A small idea of n-gram model:
Consider n=4 and there are four tokens a1, a2, a3 and a4,

$$P(a1\ a2\ a3\ a4\ in\ the\ new\ code) = \frac{count(a1\ a2\ a3\ a4)}{count(a1\ a2\ a3\ *)}$$

That is, the ratio of "number of times a4 follows a1 a2 a3 sequence" to the "number of times some other token follows a1 a2 a3 sequence".

Hindle et. Al. [8] were the first ones to view a programming language as a natural language. They have experimented and found out that code can be usefully modeled by statistical language models and such models can be leveraged usefully for software engineers. They provide a validation that any programming language consists of very predictable repetitive sequence of tokens, like how the English language is, i.e. a programming language is also natural. The language model used is n-gram model. Their work uses a code corpus to estimate the probability distribution of code sequences. The cross entropy of their model decreased rapidly with the order of n-grams, and saturated at n value of 3 to 4. They also observed that there is more amount of local regularity in software, than that in English, thus showing the high degree of naturalness in programming languages. Meanwhile, Rosenfeld came up with a Bayesian technique for statistical language modeling [2].

Following Hindle et. Al., Allamanis et. Al. [9] use the same n-grams, but using a new data set - a new curated corpus of 14,807 open source Java projects from GitHub, comprising over 350 million lines of code (LOC). The first Giga token probabilistic language model of source code is provided. The benefit of the large corpus is that, n-gram models are "data hungry", that is, adding more data almost always improves their performance.

They also introduce a new model called collapsed model. In this model, from the training data, all the identifier tokens are removed and marked as a single identifier. Now the tokenized corpus contains only tokens such as keywords (e.g. if, for, throws, and) and symbols (e.g. f, +) that represent the structural side of code. This means that the model is expected to predict only that an identifier will be used, but not which one. Using this tokenization, the cross entropy achieved is much lower than the one achieved by the full model. This signifies the importance of the identifiers when learning and understanding code: Identifiers and literals are mostly responsible for the unpredictability of the code.

But, the n-grams technique fails when the vocabulary becomes large. And n-grams saturates after observing a fraction of the data, i.e., it is specific to the training set used. But, in the real scenario, software repositories are massive depots of unstructured data, so good models require a lot of capacity to be able to learn from the voluminous scale. Software artifacts are laden with semantics, which means approaches that depend on matching lexemes are suboptimal. Practical Software Engineering tasks require a lot of context—much more than short lists of the last two, three, and four terms in a sequence. Lastly, language models, including software language models, based on n-grams are quickly overwhelmed by the curse of dimensionality, so the effective amount of context is limited.

Hence, it is not a very good idea to use software language modeling for the purpose of learning programs.

2.3 NEURAL NETWORKS IN LEARNING SOFTWARE LANGUAGES

As an improvement over the statistical language models, Bengio et al. [13] proposed a statistical model of natural language based on neural networks to learn distributed representations for words to allay the curse of dimensionality. They experimented and observed that one training sentence increases the probability of a combinatorial number of similar sentences.

To overcome the fallacies of n-grams, White et. Al. [12] proposed a RNN, which is capable of capturing long term dependencies. Their work is the pioneer work in applying deep learning for Software language modeling. Unlike Bengio et. Al., they have not used statistical modeling at all. Their model is completely based on learning. They claim that using Recursive Neural Network is not an indiscriminate introduction of complexity, but it will yield tremendous advances in the field of Software Engineering tasks. Their model achieved a cross entropy of the order of 2 bits.

A comparative study of the various neural networks (Feed forward, Recurrent and Long Short Term Memory) is done by Sundermeyer et. Al. [11]. Their work statistically analyses the pros and cons of the three different neural network models. One of the major take-aways of the work is the tremendous advantage of using LSTM Networks. Recursive Neural Networks, unlike Feedforward networks do not depend on a predefined number of tokens to be taken into the sequence. Instead, they have the capability to capture any number of token sequences that come into the current context.

While the RNN architecture itself facilitates the use of long range history information, it was found that standard gradient-based training algorithms fall short of learning RNN weight parameters in such a way that long-range dependences can be exploited. This is due to the fact that, as dependences get longer, the gradient calculation becomes increasingly unstable: Gradient values can either blow up or decay exponentially with increasing context lengths. As a solution to this problem, a novel RNN architecture was developed which avoids vanishing and exploding gradients, while it can be trained with conventional RNN learning algorithms. This idea was subsequently extended and the resulting improved RNN architecture is referred to as Long Short-Term Memory Neural Network (LSTM).The neural network architecture for LSTMs can be chosen exactly as RNN architecture, except that the standard recurrent hidden layer is replaced with an LSTM layer instead.

2.4 OTHER WORKS RELATED TO LEARNING IN SOFTWARE LANGUAGES

Kagdi et. Al. [3] have made an attempt to mine programs for function call usage patterns. A function-call-usage pattern is a list or set of function calls found in the source code. Mining call-usage patterns from source code can be considered as an instance of the general problem of frequent-pattern mining from any type of data. In this paper, two methods for mining function-call usage patterns are compared.

Itemset mining and Sequential pattern mining. The input data to frequent-pattern mining algorithms are in the form of transactions (e.g., customer baskets or items checked-out together in market-basket analysis). Here, an individual transaction corresponds to a single function definition. The support of a pattern is the number of transactions in which it occurs i.e., the number of functions in which it appears. A frequent pattern has a support at or above that of a user-specified minimum support in the considered dataset. Such frequent patterns are typically used to form association rules between a pair of patterns (e.g., when pattern A occurs, pattern B also occurs). The confidence of a rule is used to determine the strength of an association rule, and is generally computed from the support of the two patterns to a value in the range [0, 1.0]. A high confidence for a rule means the two patterns that make up the rule co-occur in most transactions.

2.5 CODE CLONE DETECTION SYSTEMS

Code clone detection systems greatly help in understanding the features that contribute to the semantic similarity of different programs.

Flavius et. Al [10] suggested that the semantic similarity between programs can be captured using the Abstract Syntax Tree. Two programs are compared for similarity by the comparison of every sub tree of AST of one program to every sub tree of the AST of the other. The degree of similarity is given by the number of sub trees that exactly match. Shruti Jadon [16] improved the former method by adding an SVM classification engine along with the exact match of AST. Mere comparison of ASTs might miss out on some cases of tricky code plagiarism, for instance, changing a loop from *for* to *while.* Hence, an SVM classification engine is trained with such tricky cases and then used for classifying whether two sub trees are similar or not. Sheneamer et. Al. [17] applied machine learning as well, and made use of the AST and the Program Dependency Graph (PDG) as features. Ashish et. Al. [18] tried to cluster similar code snippets using K-Means clustering.

Jonsson et. Al. [7] used a different technique for clone detection. They used three methods, sliding window over characters, creation of suffix trees from the code, and comparing the ASTs of the codes. They also made use of Directed Acyclic Graph (DAG) to capture the similarity between codes.

Bellon et. Al. [4] compared the above methods of clone detection and conclude that AST based techniques perform better than PDG based techniques.

Yang Yuan et. Al. [5] came up with a totally different method of identifying code clones. The usage details of all the variables are captured in a count vector. The count vectors of all the variables of a program form the distinct count matrix of the program. The degree of similarity between two programs is given by the number of vectors that are equivalent in the count matrices of the two programs.

CHAPTER 3
PROPOSED MODEL –DEEP LEARNING OF THE PROGRAM THROUGH ABSTRACT SYNTAX TREE

3.1 OVERVIEW

The entire system can be viewed as three parts. The IDE through which the user types the code and interacts with the system, the core statement prediction engine, and the data which is used for training the neural network.

The IDE serves as a tool for the teacher to add new problems along with their description and test cases. The student is displayed with all the problems available and chooses the problem to be solved. As and when the student types the code, the incomplete program is converted into an AST and subsequently to a numerical sequence, which is fed to the RNN-LSTM in order to predict the next probable node. If the predicted node and the node given by the user do not match, then the system assumes that the node produced as a result of the user's current statement is wrong. And hence, the user's current statement is wrong.

The statement prediction engine goes through two major phases. The training phase and the prediction phase. Figure 3.1 shows the overall architecture of the system. The red arrows indicate flow of data in the training phase and the black lines show the flow of data in the prediction phase. The user-typed code, once found as the correct program for the problem, is added to the training data.

In order to feed the program into the LSTM-RNN, the program has to be represented numerically. For this purpose, the program is converted into an Abstract Syntax Tree. Each node in the AST is a type of a programming construct and is mapped to a number. By feeding a sequence of these nodes, the network learns the type of the next node that comes in the sequence. The algorithm used for learning is Back propagation through time. In this method, the network learns the next probable type of programming construct that must come after the current sequence of programming constructs used.

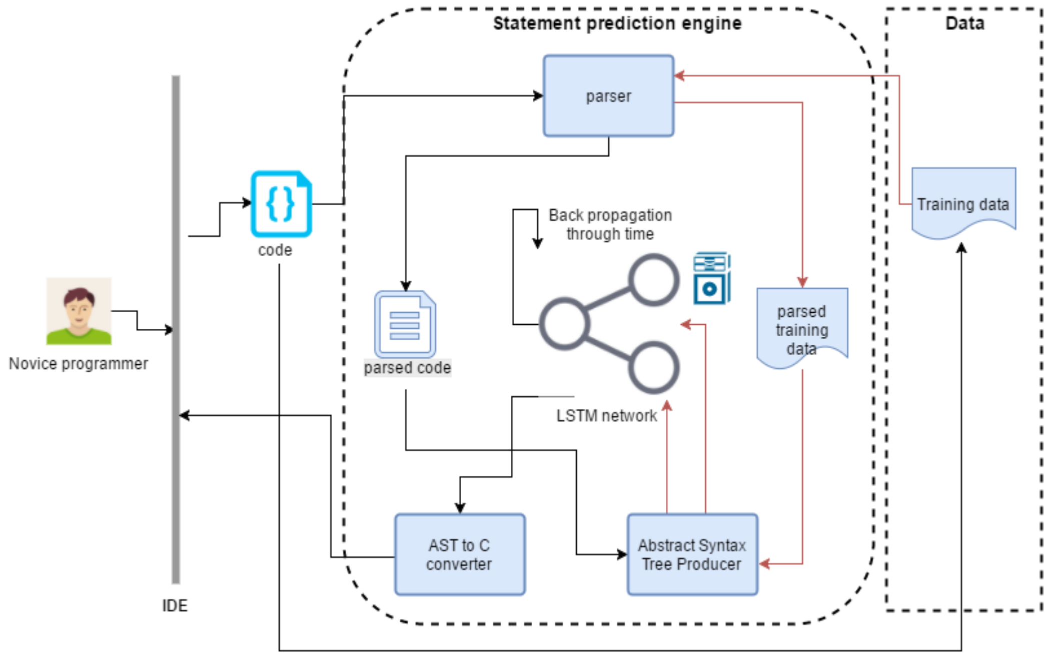

Figure 3.1 System Architecture

After the student completes the code, the solution is submitted and verified against the test cases given by the teacher for the problem solved. If the test cases are verified, the solution is taken to be correct solution and is added to the training set. The RNN-LSTM network is trained with this new solution.

3.2 MODULES

3.2.1 Preprocessing

The input sequence of statements has to be parsed to remove comments and preprocessed to expand macros. Comments and macros are solely for the programmer’s readability. Macros make coding easier and reduce the number of lines of code, while comments help in understanding and documentation of the code. Both Macros and Comments do not make any difference in the logic and working of the program. Hence they are removed by preprocessing.

3.2.2 Production of Abstract Syntax Tree

The parsed program is converted into an AST in order to feed it into the Long Short-Term Memory Neural Network. The nodes in the resultant AST are classified based on their types. This indirectly classifies every element in the program into a type of programming construct. The program is converted into a sequence of such classes and fed into the LSTM-RNN. This will learn the type of the next probable node for the given sequence during training, and predict the same during prediction phase.

3.2.3 Training and Prediction using LSTM

The RNN with LSTM is used to learn the numerical sequences of programs. The trainer module trains the LSTM-RNN network whenever a new correct program is fed to the system, and the predictor module predicts the next statement in the context of the problem to solved, using the trained LSTM-RNN. Figure 3.2.a and Figure 3.2.b show the training and testing modules of the system.

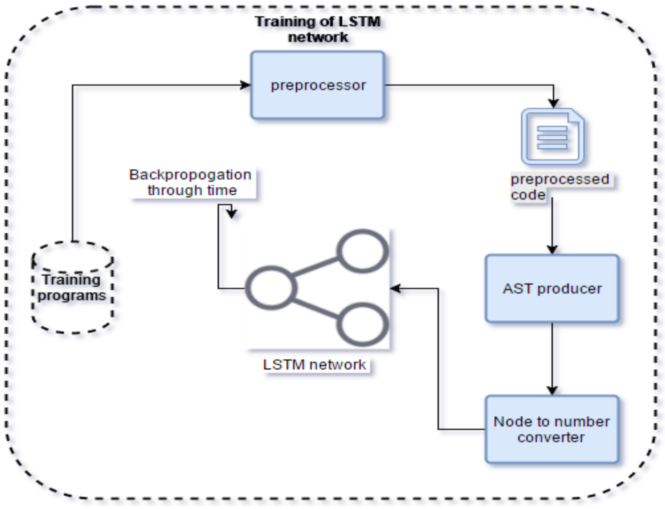

Figure 3.2.a Training Phase of RNN-LSTM

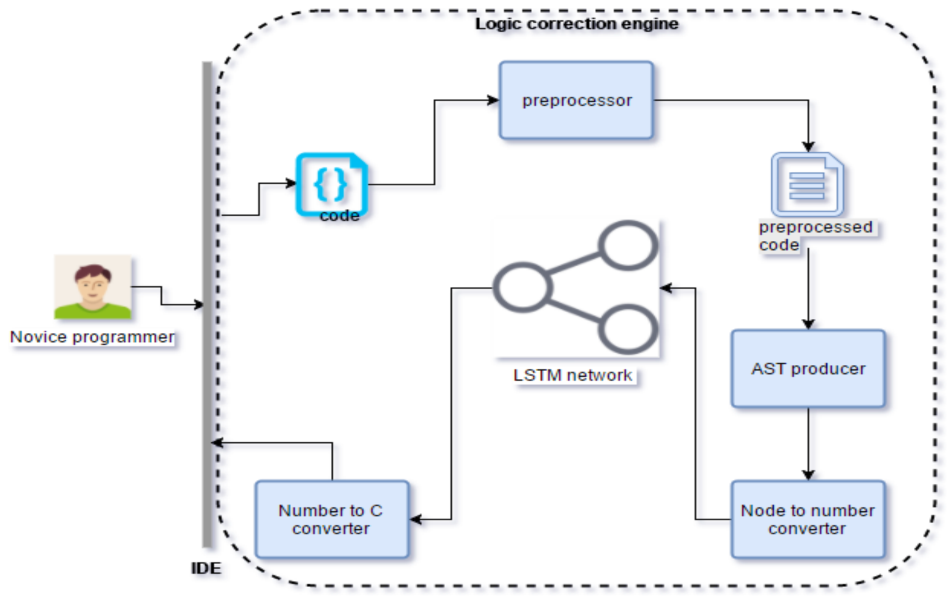

Figure 3.2.b Prediction Phase of RNN-LSTM

3.3.4 Variable Prediction

Different programs for the same problem, contain variables of different names, but doing the same purpose. These “same purpose, different name” variables have to be identified and replaced by a common identification.

e.g., Two programs for matrix addition, have the variable names <r, c> and <m, n> for denoting the number of rows and columns of the matrix. The variables in all the programs that are used for the number of rows and columns should be renamed with the common identifiers <id1, id2>.

The variable prediction module identifies the usage of variables that will lead to incorrect solution for the problem. For eg., consider the following code.

```
int i, ,j, n;

n=10;

for(i=0; i<n; i++)
```

```
{
        for(j=0; j<n; j++)
        {
                printf("%d%d\n", i, j);
        }
}
```

Here, if the user types

```
        for ( i=0;  i<j;  i++)
```

instead,

then the system should predict the error and suggest n instead of j.

CHAPTER 4

IMPLEMENTATION

4.1 PREPROCESSING

The input sequence of statements has to be parsed to remove comments and combine the declaration statements of the same type into a single statement. The preprocessing is done using the gcc preprocessor, which removes the comments, substitutes the macros and makes the program ready for parsing.

"gcc<file name> -E -std=c99 -I utils/fake_libc_include>preprocessed.c"

The preprocessed code is then parsed using the pycparser python module. It uses the Lex and Yacc to parse the preprocessed code. An outline of this module is shown in the figure 4.1.

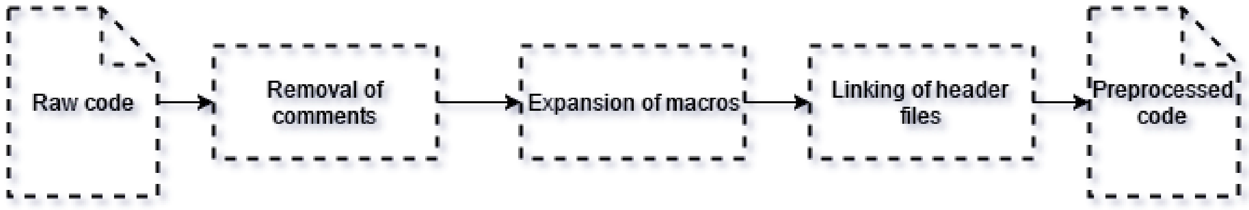

Figure 4.1 Parser Module

4.2 PRODUCTION OF ABSTRACT SYNTAX TREE

The preprocessed C code is parsed and converted into an AST. The nodes in the resultant AST are classified based on their types. There are a total of 47 different types of nodes possible in an AST for C language. Each of these types are numbered and hence the AST is converted into a tree of integer values. A preorder traversal of the tree gives us a sequence of numbers that is suitable to be fed into the RNN-LSTM network. Figure 4.2 shows the flow of the AST module.

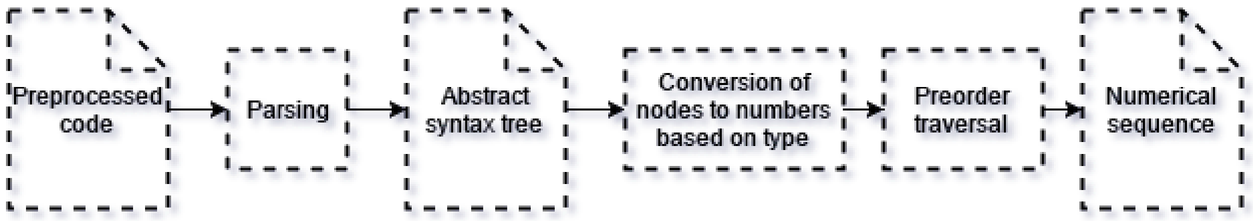

Figure 4.2 Flow of Conversion of C Code into Numerical Sequence.

The Abstract Syntax Tree production module is implemented using the c_ast module of Python. An abstract algorithm used for the preorder traversal of the AST is given in Algorithm 1.

Algorithm 1: Conversion of AST into numerical sequence

Data: One node of the Abstract syntax tree.

Result: Sequence of numbers

```
1: type :=get_type(current_node)
2: sequence.append(get_number(type))
3: if type = operator
4:     sequence.append(get_number(type.value))
5: if current_node has children
6:     for child in current_node.children
7:     goto1 with child  ascurrent_node
8: end
```

The implementation is executed on 10 different programs for finding whether a given number is prime or not. It is also executed for an incomplete code.

4.3 ABSTRACT SYNTAX TREE NODES

The Abstract Syntax Tree nodes for C language can be classified into 47 different classes. Some of those node types are shown in the Table 4.1. The classification of the nodes is language specific. The attributes with * are children to the node.

Table 4.1 Types of Nodes in Abstract Syntax Tree for C Language

Node	**Attributes**
ArrayDecl	[type*, dim*, dim_quals]
BinaryOp	[op, left*, right*]
Constant	[type, value]
ID	[name]
If	[cond*, iftrue*, iffalse*]

4.4 EXTENSION OF THE ABSTRACT SYNTAX TREE NODES FOR PREDICTION OF OPERATORS

The RNN-LSTM predicts the next possible AST node that is likely to follow the given sequence of nodes. With the node types mentioned in Table 4.1, the prediction will be restricted to “unary operator” and “binary operator” when it comes to operators. This is a drawback because, what operator is to be used is very important and operators make a big difference in the solution of a problem. Hence, the node types list is extended by including all the unary and binary operators. So, the number of classes increases to 67 from 47.

4.5 DESIGN OF RNN-LSTM NETWORK

The designing of the Neural network involved the use of the tool “TensorFlow”. TensorFlow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

Using the RNN-LSTM module in TensorFlow, a training and prediction system specific to our application is developed with 600 units of hidden LSTM cells. TensorFlow uses the RNN-LSTM cell which works ideally for sequence to sequence model with attention mechanism [13]. The model also uses a softmax layer at the output layer as referred in [14]. The reason for using softmax is to convert the outputs into probabilities of each class being the next node. The graphical representation of the neural network that is developed is shown in Figure 4.3.

4.6 TRAINING PHASE

The training data is divided into input and output separately and fed into the Neural network. The data available for training is a single sequence of the correct code. This single sequence of n elements is split into n-1 different sequences, with ith sequence having i elements as input and i+1 th element as output. This is done to ensure that the neural network is trained with all possible partial sequences of the code. Algorithm 2 shows the abstract algorithm for splitting the numerical sequence of correct code into the dataset of required format.

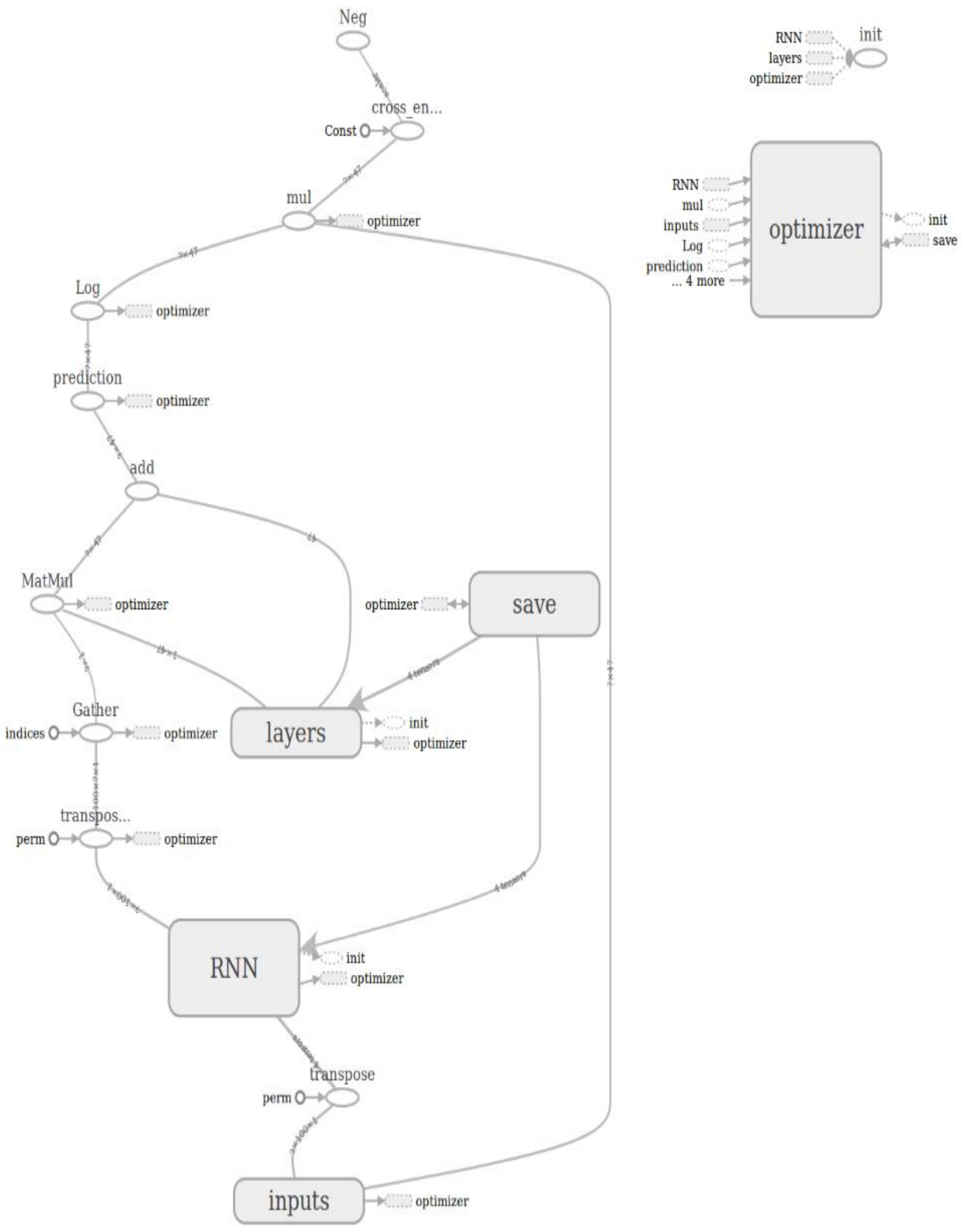

Figure 4.3 Graphical Representation of the Designed RNN-LSTM

When training the model for a new problem for the first time, the network initially has arbitrary weights. Every prediction of the Network for an input is compared to the actual output. The difference between predicted output and actual output is the error value. This error value is propagated backwards and the weights and biases are altered to make the network predict the actual value. This process is repeated for every training set and the weights are corrected accordingly to make the network efficient. The corrected weights are stored to disk.

Algorithm 2: Splitting numerical sequence into n sequences

Data: Sequence[*n*]

Output: *n* Sequences of increasing length, *n* outputs

```
1:  newSequence[n][]
2:  output[n]
3:  for i=1 to n-1
4:  newSequence[i] = Sequence[1:i]
5:       Output[i] = Sequence[i+1]
```

If the training module witnesses stored weights for this problem, the network loads the stored weights and further gets trained for the new data that is received. Figure 4.4 shows the flow of data throughout the training phase.

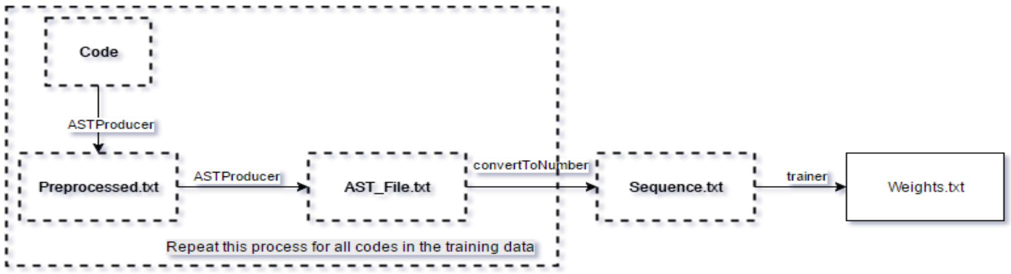

Figure 4.4 Flow of Data through the Training Phase

4.7 PREDICTION PHASE

The prediction module loads the stored weights into the neural network and proceeds with the prediction of node type. During prediction phase, the incomplete code, at every statement, the code is converted into AST and then to numerical sequence, and fed into the predictor that is trained. The predictor gives the probabilities of all the 67 classes being the next node. If the user's next node does not match with the top five guesses of the predictor, then an error message is given. Algorithm 3 is the abstract algorithm for prediction of next probable node. Figure 4.5 shows the updating and usage of weights by the training and prediction modules.

Algorithm 3: Prediction of next probable node

Data: User's incomplete code

Output: List of corrections to be made in the code to give logically correct solution

```
 1:  preprocess("incomplete_code.c")
 2:  ast  := ASTProducer.convert("incomplete_code.c")
 3:  sequence := numerical_sequence(ast)
 4:  nn := load(RNN-LSTM, trained_weights)
 5:  result := n.predict(sequence)
 6:  predicted := top_five(result)
 7:  corrections = []
 8:  if( sequence.get_last() not in predicted )
 9:      corrections.append( getName(sequence.get_last() ) + "should be replaced
     by" + getName( predicted.get_first() ) )
10:  end
```

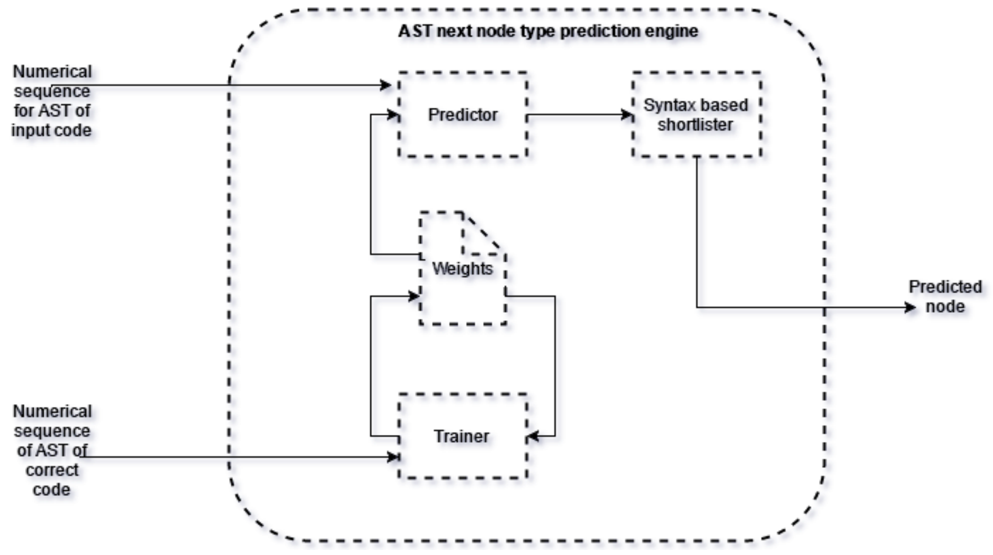
AST next node type prediction engine
Numerical sequence for AST of input code
Predictor
Syntax based shortlister
Weights
Predicted node
Numerical sequence of AST of correct code
Trainer

Figure 4.5 Update and Usage of Weights by Training and Prediction Modules

4.8 SYNTAX BASED SHORTLISTING OF NODE TYPES

During prediction, the RNN gives the probability of each of the 46 types to follow the given sequence. The one with the highest probability is chosen to be the predicted node. Instead of considering the probabilities of all the 46 nodes, only those that can syntactically follow the previous node type are considered. This increases the prediction accuracy since the choices are narrowed down. Algorithm 4 shows the algorithm for prediction using shortlisting of nodes based on syntax.

Algorithm 4: Syntax based shortlisting of predicted nodes

Data: Probabilities of 46 nodes

Output: Predicted node

```
1:  isValid[46][46]
2:  for i=1 to 46
3:  for j=1 to 46
4:  if ( j following i is syntactically correct )
```

```
5:  isValid[i][j] = true
6:  max := 0
7:  predicted := 0
8:  prev := previousNodeType
9:  for i=1 to 46
10: if ( prob[i] >max and isValid[prev][i])
11: max:=prob[i]
12: predicted := i
13: return predicted
```

4.9 VARIABLE PREDICTION USING COUNT VECTOR

The prediction system using the Abstract Syntax Tree assumes all variables to be equal. Hence it calls all the variables with the same name "ID". Both i<j and j<i are interpreted as ID<ID. This flaw is overcome by the use of count vectors. At each node of the AST, along with the information of the node type, the characteristic vector determining the variable uniquely, is stored and used for prediction. The features of the count vector of identifier are listed in Table 4.2

Table 4.2 Features of the Count Vector of an Identifier

S NO	NAME OF THE FEATURE TO BE COUNTED
1	Used
2	Added or subtracted
3	Multiplied or divided
4	Invoked as a parameter
5	In an if statement
6	As an array subscript
7	Defined

8	Defined by add or subtract operation
9	Defined by multiply or divide operation
10	Defined by an expression which has constants in it
11	In a third level loop (or deeper)
12	In a second level loop
13	In a first level loop

The count of these features forms the count vector. The count vector of every variable at every usage is recorded in the AST. These features are learnt along with the AST. Whenever the user types a variable whose feature does not match the feature learnt by the predictor, the system raises an error.

4.10 RESULTS

The network is trained with 45 programs for sum of digits of a number problem.

The model is tested for various learning rates. The learning rate determines how much the weights are to be penalized for every wrong prediction of the model during training. An Epoch is the number of times the same training data is fed into the RNN to train the network. Figure 4.6.a, 4.6.b, 4.6.c, 4.6.d and 4.6.e show the graphical representation of variation of loss rate and accuracy with every epoch for learning rates 0.0001, 0.001, 0.0003, 0.0005 and 0.0009 for the values shown in Table 4.3. It can be inferred from the graphs that as we decrease the learning rate, there is an impact in the randomness of the prediction. This is because, with higher learning rates, there is higher penalization of weights and hence the weights are altered to a greater extent. Because of this there is a drastic change in the forthcoming predictions. Another inference from the graph is that, with decreasing learning rates, the number of epochs required in training for the complete reduction of loss rate increases. Since penalization of weights is very less, number of steps taken is more.

Table 4.3 Variation of Accuracy and Loss Rate with every Epoch for Various Learning Rates

Learning Rate	0.0001		0.0003		0.0005		0.0009		0.001	
Epoch	**Accu-racy**	**Loss Rate**	**Accu-Racy**	**Loss Rate**	**Accu-racy**	**Loss Rate**	**Accu-racy**	**Loss Rate**	**Accu-racy**	**Loss Rate**
1	26.23	80.33	34.43	78.69	29.51	73.77	39.34	72.13	26.23	78.69
40	85.25	59.02	86.89	65.57	80.33	65.57	83.61	65.57	78.69	63.93
80	88.52	49.18	80.33	54.10	91.80	47.54	85.25	50.82	83.61	57.38
120	96.72	26.23	98.36	16.39	98.36	27.87	91.80	44.26	95.08	45.90
146	100.00	16.39	100.00	0.00	93.44	37.70	100.00	26.23	100.00	18.03
186	100.00	1.64	100.00	0.00	98.36	6.56	100.00	14.75	100.00	0.00
193	100.00	1.64	100.00	0.00	100.00	0.00	100.00	8.20	100.00	0.00
224	100.00	1.64	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00
228	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00

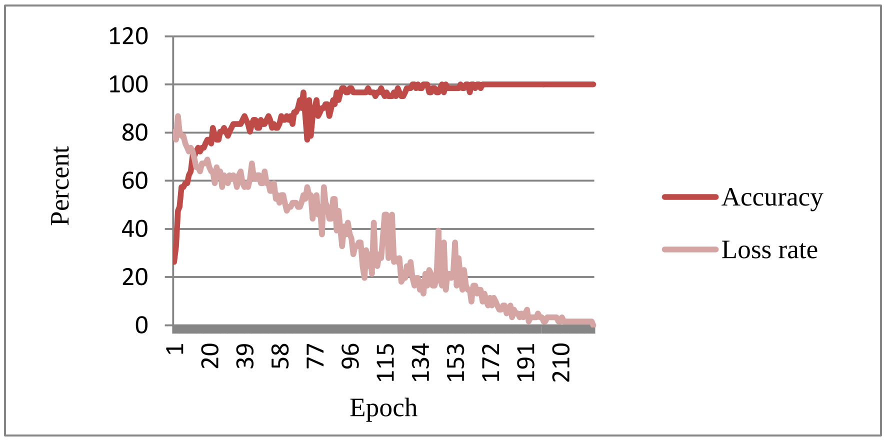

Figure 4.6.a Variation of Accuracy and Loss Rate with every Epoch for Learning Rate 0.0001

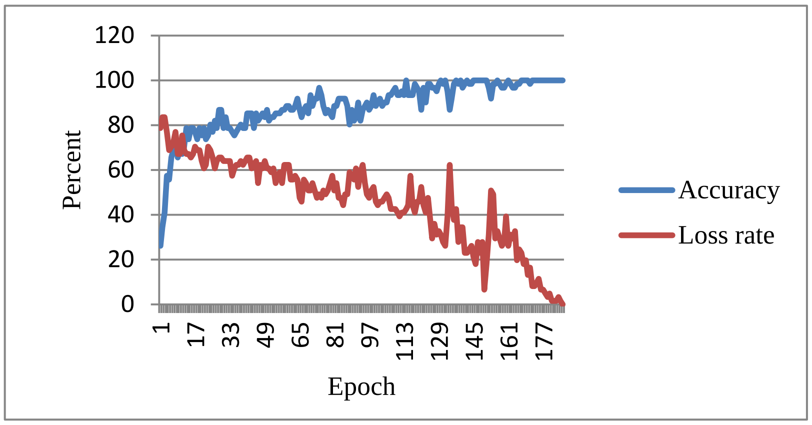

Figure 4.6.b Variation of Accuracy and Loss Rate with every Epoch for Learning Rate 0.001

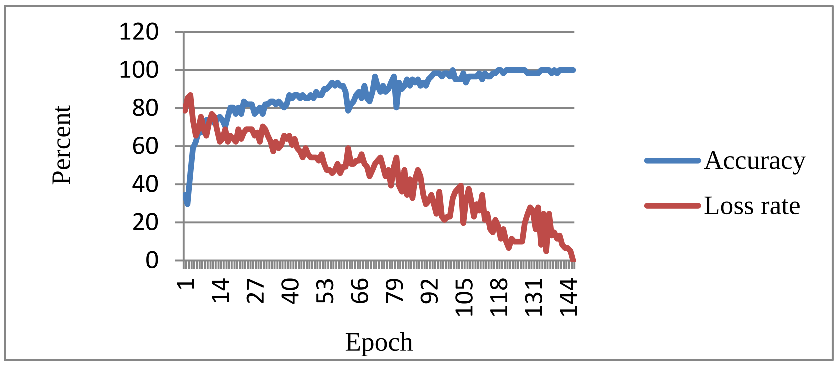

Figure 4.6.c Variation of Accuracy and Loss Rate with every Epoch for Learning Rate 0.0003

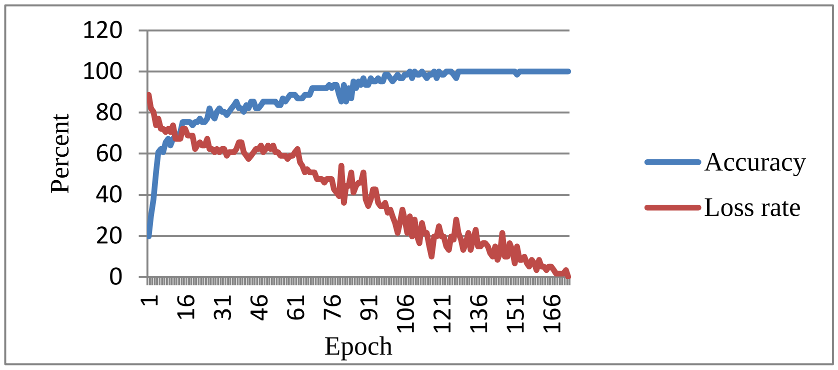

Figure 4.6.d Variation of Accuracy and Loss Rate with every Epoch for Learning Rate 0.0005

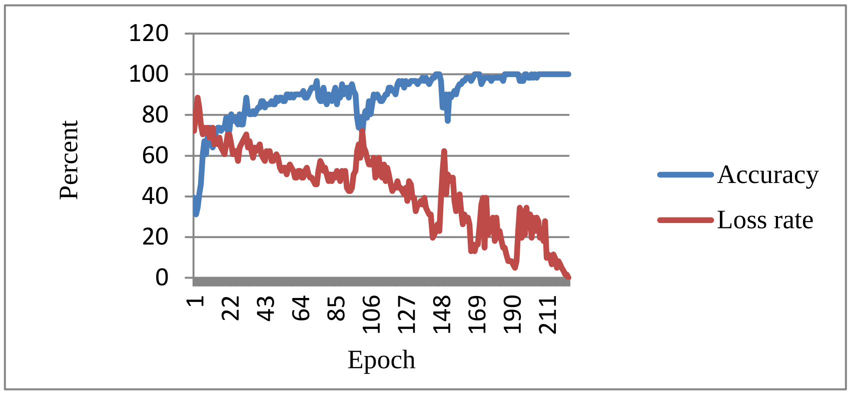

Figure 4.6.e Variation of Accuracy and Loss Rate with every Epoch for Learning Rate 0.0009

Figure 4.7.a, 4.7.b, 4.7.c and 4.7.d show the graphical representation of the variation of accuracy and loss rate during training of models with 100, 200, 450 and 600 hidden units for the values in the table 4.4. It can be inferred from the graphs that as the number of hidden units increases, the required accuracy is achieved in a very small number of training epochs.

The system is tested with functionally correct test programs for the same. Accuracy is calculated using equation 5.1.

$$a = \frac{n}{t} * 100 \tag{5.1}$$

Where,

a = accuracy,

n = number of nodes predicted correctly,

t = total number of nodes in the AST of test program.

An accuracy of 85.72% is achieved for the problem of sum of digits of a number. The error of 15% is due to false positives i.e., predicting the statement to be wrong even when it is right.

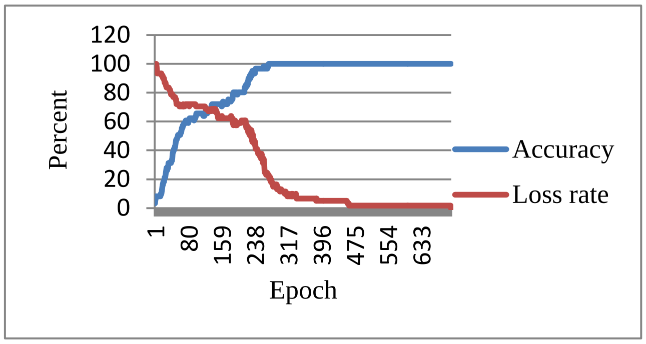

Figure 4.7.a Variation of Accuracy and Loss Rate with every Epoch for 100 Hidden Units

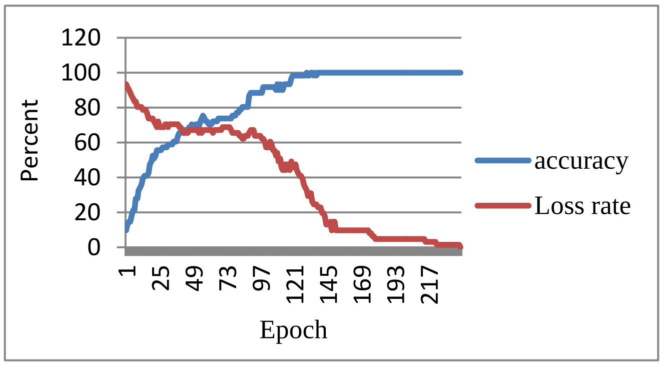

Figure 4.7.b Variation of Accuracy and Loss Rate with every Epoch for 100 Hidden Units

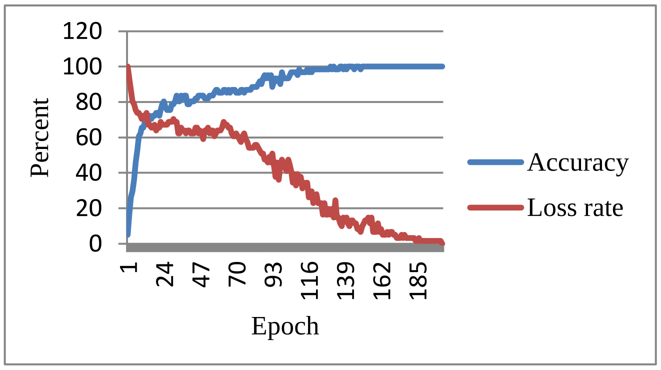

Figure 4.7.c Variation of Accuracy and Loss Rate with every Epoch for 450 Hidden Units

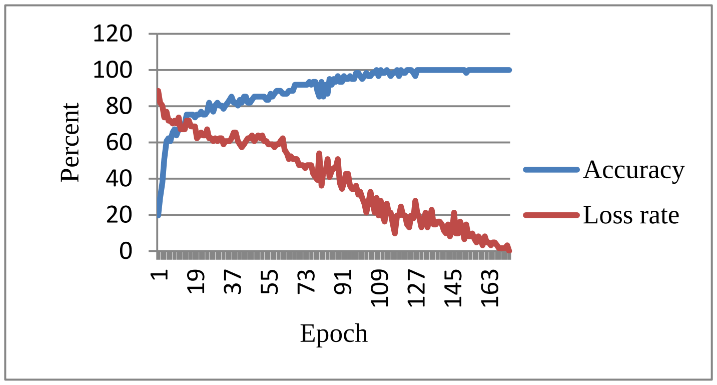

Figure 4.7.d Variation of Accuracy and Loss Rate with every Epoch for 600 Hidden Units

Table 4.4 Variation of Accuracy and Loss Rate with every Epoch for various number of Hidden Units

Hidden units	100		200		450		600	
Epoch	**Accu-racy**	**Loss Rate**	**Accu-racy**	**Loss Rate**	**Accu-racy**	**Loss Rate**	**Accu-racy**	**Loss Rate**
1	3.28	98.36	9.84	93.44	4.92	100.00	19.67	88.52
40	32.79	78.69	65.57	68.85	78.69	63.93	80.33	60.66
80	59.02	72.13	77.05	65.57	88.52	54.10	85.25	54.10
120	65.57	70.49	98.36	45.90	98.36	24.59	100.00	24.59
170	72.13	62.30	100.00	9.84	100.00	4.92	100.00	0.00
201	80.33	59.02	100.00	4.92	100.00	0.00	100.00	0.00
240	96.72	40.98	100.00	0.00	100.00	0.00	100.00	0.00
704	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00

CHAPTER 5

CONCLUSION AND FUTURE WORK

The system learns to check for the right solution for the “sum of digits of a number” problem by learning various solutions for the same problem. An accuracy of 85% is achieved and the 15% error is mainly contributed by the false positives.

It is language dependent and the same system can be implemented for various other languages by modifying the types of nodes used in the abstract syntax tree.

There is a lot of scope for future work in the project. The tagline here is used to train the network for the given program. This can be found automatically by applying learning over the identifier names in the program. (The identifier names are meaningful and often relate to the program). This system is similar to a tutor. Hence, it can be made user specific. The system can be made to identify the potential mistakes that are frequently made by a specific user, and train that user to work towards reducing the regularly made mistake.

REFERENCES

[1] Sepp Hochreiter and Jurgen Schmidhuber, “Long Short-Term Memory”, IEEE Neural Computation, Vol. 9, No. 8, pp. 1735 - 1780, 1997.

[2] R. Rosenfeld, “Two Decades of Statistical Language Modeling: Where Do We Go From Here?”, Proc. IEEE, Vol. 88, No. 8, pp. 359–394, 2000.

[3] Huzefa Kagdi and Michael L. Collard and Jonathan I. Maletic, “Comparing Approaches to Mining Source Code for Call-Usage Patterns”, Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops), IEEE, pp. 20 - 20, 2007.

[4] Stefan Bellon and Rainer Koschke and Giulio Antoniol and Jens Krinke and Ettore Merlo, “Comparison and Evaluation of Clone Detection Tools”, IEEE Transactions on Software Engineering, IEEE, Vol. 33, No. 9, pp. 577 – 591, 2007.

[5] Yang Yuan and Yao Guo, “CMCD: Count Matrix Based Code Clone Detection”, 18th Asia-Pacific Software Engineering Conference, IEEE, pp. 250-257, 2011.

[6] T. Mikolov and S. Kombrink and L. Burget and J. Cernock´y, and S. Khudanpur, “Extensions of Recurrent Neural Network Language Model”, in Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP, IEEE Signal Processing Society, pp.5528–5531, 2011.

[7] Mikkel Jonsson Thomsen and Fritz Henglein, “Clone Detection using Rolling Hashing, Suffix Trees and Dagification: A Case Study”, 6th International Workshop on Software Clones (IWSC), IEEE, pp. 22-28, 2012.

[8] Abram Hindle and Earl T. Barr and Zhendong Su and Mark Gabel and Premkumar Devanbu, “On the Naturalness of Software”, 34th International Conference on Software Engineering (ICSE), pp. 837-847, 2012.

[9] Miltiadis Allamanis and Charles Sutton, “Mining Source Code Repositories at Massive Scale using Language Modeling”, 10th Working Conference on Mining Software Repositories (MSR), pp. 207 – 216, 2013.

[10] Flavius Mihai Lazar and Ovidiu Banias, “Clone Detection Algorithm Based on the Abstract Syntax Tree approach”, 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), IEEE, pp. 73-78, 2014.

[11] Martin Sundermeyer and Hermann Ney and Ralf Schlüter, “From Feedforward to Recurrent LSTM Neural Networks for Language Modeling”, IEEE/ACM Transactions on Audio, Speech, and Language, Vol. 23, No. 3, pp. 517-529, 2015.

[12] Martin White and Christopher Vendome and Mario Linares-Vasquez and Denys Poshyvanyk, “Toward Deep Learning Software Repositories”, IEEE/ACM 12th Working Conference on Mining Software Repositories, pp. 334 – 345, 2015.

[13] Sébastien Jean and Kyunghyun Cho and Roland Memisevic and Yoshua Bengio, “On Using Very Large Target Vocabulary for Neural Machine

Translation", arXiv, Cornell University Library, Vol. 2, pp. 2007 – 2017, 2015.

[14] Oriol Vinyals and Lukasz Kaiser and Terry Koo and Slav Petrov and Ilya Sutskever and Geoffrey Hinton, "Grammar as a Foreign Language", arXiv, Cornell University Library, Vol. 3, pp.7449 – 7459, 2015.

[15] Hamid Palangi and Li Deng and YelongShen and JianfengGao and Xiaodong He and Jianshu Chen and Xinying Song and Rabab Ward, "Deep Sentence Embedding Using Long Short-Term Memory Networks: Analysis and Application to Information Retrieval", IEEE/ACM Transactions on Audio, Speech, and Language , Vol. 24, No.4, pp. 694-707, 2016.

[16] Shruti Jadon, "Code Clones Detection using Machine Learning Technique: Support Vector Machine", International Conference on Computing, Communication and Automation (ICCCA), IEEE, pp. 399 – 303, 2016.

[17] Abdullah Sheneamer and Jugal Kalita, "Semantic Clone Detection Using Machine Learning", 15th IEEE International Conference on Machine Learning and Applications (ICMLA), IEEE, pp. 1024-1028, 2016.

[18] Aveg Ashish, "Clones Clustering using K-means", 2016 10th International Conference on Intelligent Systems and Control (ISCO), pp. 1-6, 2016.