
lexical Analyzer

By

Tunazzinur Rahman Kabbo ID:19202103268

Md. Zobayer Hasan Nayem ID:19202103274

Afrina Akhter Mim ID:19202103310

Submitted in partial fulfillment of the requirements of the degree of

**Bachelor of Science in
Computer Science and Engineering**



Department of Computer Science and Engineering
Bangladesh University of Business and Technology

November 2022

Declaration

We do hereby declare that the project works presented here with entitled as, “Lexical Analyzer” are the results of our own works. We further declare that the project has been compiled and written by us and no part of this project has been submitted elsewhere for the requirements of any degree, award or diploma or any other purposes except for this project. The materials that are obtained from other sources are duly acknowledged in this project.

Signature of Developers

Tunazzinur Rahman Kabbo, Id: 19202103268

Md. Zobayer Hasan Nayem, Id: 19202103274

Afrina Akhter Mim Id: 19202103310

Approval

I do hereby declare that the project works presented here with entitled as, Lexical Analyzer, are the outcome of the original works carried out by Tunazzinur Rahman Kabbo, Md. Zobayer Hasan Nayem, Afrina Akhter Mim, under my supervision. I further declare that no part of this project has been submitted elsewhere for the requirements of any degree, award or diploma or any other purposes except for this project. I further certify that the dissertation meets the requirements and standard for the degree of Doctor of Philosophy in Computer Science and Engineering.

Chairman

Md. Saifur Rahman

Assistant Professor

Department of Computer Science Engineering

Bangladesh University of Business and Technology

Supervisor

T. M. Amir - Ul - Haque Bhuiyan

Assistant Professor

Department of Computer Science Engineering

Bangladesh University of Business and Technology

Dedication

We would like to dedicate this project to our loving parents . . .

Acknowledgement

We are deeply thankful to Bangladesh University of Business and Technology (BUBT) for providing us such a wonderful environment to peruse our project. We would like to express our sincere gratitude to T. M. Amir - Ul - Haque Bhuiyan, Assistant Professor, CSE, BUBT. We have completed our project with his help. We found the project area, topic, and problem with his suggestions. He guided us with our study, and supplied us many articles and academic resources in this area. He is patient and responsible. When we had questions and needed his help, he would always find time to meet and discuss with us no matter how busy he was. We also want to give thanks to our CSE department. Our department provide us logistic supports to complete our project with smoothly. We would also like to acknowledge our team members for supporting each other and be grateful to our university for providing this opportunity for us.

Abstract

In this report, we will not present compiler construction techniques in usual sense. Instead, a functional specification technique is discussed. The study is focused on the dynamic semantics issue – the code generation. The development is conducted in terms of constructive type theory using the mathematical theorem proof development system - PowerEpsilon. Three programming languages are investigated, TINY, C- and Pascal-, which are used as the sample languages for teaching compiler construction technologies. We are not going to give the description of compiler construction in terms of a general-purpose programming language such as Pascal, C or Ada. Instead, what we provide are the abstract specification of compiler construction process and algorithm. So it is where the name of the book come from. Also the compilation techniques we are going to specify are focused on the imperative programming languages only. The following programming languages will not be investigated: - The functional programming languages such as LISP and ML; - The logic programming languages such as Prolog; - The object-oriented programming languages such as C++, Smalltalk and Java; - The concurrent programming languages such as Ada; - The synchronous dataflow languages such as SCADE/LUSTRE. The compiling process is described as a state transformation with a continuation based method. Therefore, this compiler specification technique is very similar to the denotational semantic technique used for specifying the semantics of programming languages. This work is a little different from our previous work where the semantics of source language and the semantics of target language applying to the same abstract states. In this work, the semantics of source language and the semantics of target language are described in terms of different abstract states. We have also investigated the compilation techniques for a programming language with procedures and stack-based run-time environment.

Contents

Declaration	iii
Approval	iv
Dedication	v
Acknowledgement	vi
Abstract	vii
List of Figures	x
1 Introduction	1
1.1 Introduction	1
1.2 Problem Statement :	2
1.3 Problem Background :	2
1.4 Project Objective :	4
1.5 Motivations	4
1.6 Project Contribution	4
1.7 Project Report Organization	5
1.8 Summary	5
2 Literature Review	6
2.1 Introduction	6
2.2 Related work	6
2.3 Problem Analysis	7
2.4 Summary	8
3 Methodology	9
3.1 Introduction	9
3.2 Feasibility analysis	9
3.3 Requirement Analysis	10

3.4	Summary	11
4	Implementation and Testing	12
4.1	Introduction	12
4.2	System Setup	12
4.3	Evaluation	12
4.4	Results and Discussion	13
4.5	Summary	13
5	Conclusion	14
5.1	Conclusion	14
5.2	Limitation and Future Works	14
5.2.1	Limitation	14
5.2.2	Future Works	14
	References	24

List of Figures

1.1	Lexical Analyzer	2
1.2	Exceeding length of identifier or numeric constants.. . . .	2
1.3	Exceeding length of identifier or numeric constants.. . . .	3
1.4	Transposition of two characters...	3
4.5	User Input File	12
4.6	Compiler Code Codeblocks Editor	12
4.7	Error Output From Codeblocks Compiler	13
4.8	Without-Error Output From Codeblocks Compiler	13

1 Introduction

1.1 Introduction

Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens. Lexical Analysis can be implemented with the Deterministic finite Automata. The output is a sequence of tokens that is sent to the parser for syntax analysis. A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Now we described how many Token are scanning lexical analyzer : Lists of Token:

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

To implement a lexical analyzer by hand, it helps to start with a diagram or other description for the lexemes of each token. We can then write code to identify each occurrence of each lexeme on the input and to return information about the token identified. We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a lexical-analyzer generator and compiling those patterns into code that functions as a lexical analyzer. This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patterns, not the entire program. It also speeds up the process of implementing the lexical analyzer, since the programmer specifies the software at the very high level of patterns and relies on the generator to produce the detailed code. We show how this notation can be transformed, first into nondeterministic automata and then into deterministic automata. The latter two notations can be used as input to a "driver," that is, code which simulates these automata and uses them as a guide to determining the next token. This driver and the specification of the automaton form the nucleus of the lexical analyzer.



Figure 1.1: Lexical Analyzer

The part of Compiling a Expression

- Lexical Analyzer.
- Syntax Analyzer.
- Semantic Analyzer
- Intermediate Code Generator

1.2 Problem Statement :

When the token pattern does not match the prefix of the remaining input, the lexical analyzer gets stuck and has to recover from this state to analyze the remaining input. In simple words, a lexical error occurs when a sequence of characters does not match the pattern of any token. It typically happens during the execution of a program.

1.3 Problem Background :

Types of lexical error that can occur in a lexical analyzer are as follows:

```

} #include <iostream>
  using namespace std;
}

int main() {

    int a=2147483647 +1;
    return 0;
}
  
```

Figure 1.2: Exceeding length of identifier or numeric constants..

This is a lexical error since signed integer lies between 2,147,483,648 and 2,147,483,647.

```
#include <iostream>
using namespace std;

int main() {

    printf("Geeksforgeeks");$
    return 0;
}
```

Figure 1.3: Exceeding length of identifier or numeric constants..

This is a lexical error since an illegal character \$ appears at the end of the statement.

```
#include <iostream>
using namespace std;

int main()
{
    /* spelling of main here would be treated as an lexical
    error and won't be considered as an identifier,
    transposition of character 'i' and 'a'*/
    cout << "GFg!";
    return 0;
}
```

Figure 1.4: Transposition of two characters...

Error-recovery actions are:

1. Transpose of two adjacent characters.
2. Insert a missing character into the remaining input.
3. Replace a character with another character.
4. Delete one character from the remaining input.

1.4 Project Objective :

Following analysis objectives achieved from this analysis area unit given below:

- Scan code from a file or take user input.
- Breaks these syntax's into a series of tokens, by removing any white-space or comments in the source code.
- When the compiler compile the user input or any expression then, the lexical analyzer match user input expression for default compiler expression.

1.5 Motivations

Lexical Analyzer are compile complicated pieces of software that implement delicate algorithms. Bugs in compilers do occur and can cause incorrect executable code to be silently generated from a correct source program. In other words, a buggy compiler can insert bugs in the programs that it compiles. This phenomenon is called mis-compilation.

That's why we are developing a lexical analyzer to create a easy user define and also easy expression detected. That's why we also developing a algorithm to fetch the input and proper scanning to user input. It's help any user to detect any error.

1.6 Project Contribution

The overall contribution of the research work includes,

- Lexical Analysis is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences.
- We will implement a lexical analyzer that breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.
- We define file reading feature for extraction techniques that helps to find out best accuracy.
- Finally, we will implement the system in C++, an “intermediate-level” language.

1.7 Project Report Organization

An overview of the steps of the research is organized as follows.

Chapter 2 presents the background information and literature review on the lexical analyzer. Here, we also analyze the problem.

Chapter 3 comprises our proposed architecture in detail along with the lexical analyzer requirements, the feasibility, the methodology of the research.

Chapter 4 provides a brief discussion of the implementation, tests, and evaluations to estimate our proposed algorithm. In this chapter, we also analyze the result.

Lastly, Chapter 5 is a review of our project work, including conclusions as well as discusses the objectives for future work.

1.8 Summary

This chapter comprises a broad overview of the problem such as what are we specifically targeting, what are the purposes of our project work along with the motivation of the output of the project work. This section also represents the overall steps on which we carried out our project work.

2 Literature Review

2.1 Introduction

In recent years technology has been involved in various aspect of modern life. Everyone is trying to build a secured compiler system Android studio, Pycharm, Visual Studio Code, Codeblocks ,Netbeans and so more with the help of Developers and Problem Solver. Developers needs to a fresh and easy compiler for his project and also they also need a clean UI for his work.

2.2 Related work

For the development of programming languages that were created and in use by 1967, including Fortran, Algol, Lisp, and Simula, see [7]. For languages that were created by 1982, including C, C++, Pascal, and Smalltalk, see [I]. The GNU Compiler Collection, gcc, is a popular source of open-source compilers for C, C-t+, Fortran, Java, and other languages [2]. Phoenix is a compiler-construction toolkit that provides an integrated framework for building the program analysis, code generation, and code optimization phases of compilers discussed in this book [3]. For more information about programming language concepts, we recommend [5,6]. For more on computer architecture and how it impacts compiling, we suggest [4].

- 1. Bergin, T. J. and R. G. Gibson, History of Programming Languages, ACM Press, New York, 1996.
- 2. <http://gcc.gnu.org/>.
- 4. Hennessy, J. L. and D. A. Patterson, Computer Organization and Design: The Hardware/Software Interface, Morgan-Kaufmann, San Francisco, CA, 2004.
- 5. Scott, M. L., Programming Language Pragmatics, second edition, MorganKaufmann, San Francisco, CA, 2006.

- 6. Sethi, R., Programming Languages: Concepts and Constructs, AddisonWesley, 1996.
- 7. Wexelblat, R. L., History of Programming Languages, Academic Press, New York, 1981.

Machine and Assembly Languages. Machine languages were the first generation programming languages, followed by assembly languages. Programming in these languages was time consuming and error prone.

+ Modeling in Compiler Design. Compiler design is one of the places where theory has had the most impact on practice. Models that have been found useful include automata, grammars, regular expressions, trees, and many others.

Code Optimization. Although code cannot truly be "optimized," the science of improving the efficiency of code is both complex and very important. It is a major portion of the study of compilation.

Higher-Level Languages. As time goes on, programming languages take on progressively more of the tasks that formerly were left to the programmer, such as memory management, type-consistency checking, or parallel execution of code.

Compilers and Computer Architecture. Compiler technology influences computer architecture, as well as being influenced by the advances in architecture. Many modern innovations in architecture depend on compilers being able to extract from source programs the opportunities to use the hardware capabilities effectively.

2.3 Problem Analysis

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a

lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser. These interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

2.4 Summary

Tokens: The lexical analyzer scans the source program and produces as output a sequence of tokens, which are normally passed, one at a time to the parser. Some tokens may consist only of a token name while others may also have an associated lexical value that gives information about the particular instance of the token that has been found on the input.

Lexemes: Each time the lexical analyzer returns a token to the parser, it has an associated lexeme - the sequence of input characters that the token represents.

Buffering: Because it is often necessary to scan ahead on the input in order to see where the next lexeme ends, it is usually necessary for the lexical analyzer to buffer its input. Using a pair of buffers cyclicly and ending each buffer's contents with a sentinel that warns of its end are two techniques that accelerate the process of scanning the input.

Patterns: Each token has a pattern that describes which sequences of characters can form the lexemes corresponding to that token. The set of words, or strings of characters, that match a given pattern is called a language.

Regular Expressions. These expressions are commonly used to describe patterns. Regular expressions are built from single characters, using union, concatenation, and the Kleene closure, or any-number-of, operator.

3 Methodology

3.1 Introduction

This chapter represents the proposed model and illustrates the feasibility analysis, We develop a algorithm that's work any editors plat-form. We develop this algorithm in C programming language, it's work properly, it also help to detect code error any user input.

3.2 Feasibility analysis

Feasibility analysis is the method of concluding the fallibleness of a system. This study is essential to open new concepts that could effectively improve a project's scope. So, it's best to make these decisions in advance. Now, in this part, whether the system is feasible for development or not will be discussed. This study also includes the availability of resources, evaluation of cost, how this system can benefit an organization, how the system can maintain after developed. There are four types of feasibility to measure this analysis that is technical, economical, operational, and schedule.

Technical Feasibility: In technical feasibility, whether we have the technical knowledge to manage the completion of the research has been discussed. Mainly, how hardware and software meet our proposed system has been evaluated in this part. To complete this research, we will use Codeblocks IDE. File browser, intelligent auto-completion, run in IDE with input & console windows for trial and error development are the features of Codeblocks. And for hardware, the configuration has to be minimum RAM 4 GB, hard disk space 1.5 GB + caches at least 1 GB. So, it can be said using this software and hardware, the proposed system can be developed efficiently.

Economic Feasibility: In economic feasibility, the cost of developing our system has been analyzed. Also, define whether the system is capable of producing financial gains for an organization. The cost of completing our research which is the recognize the digit don't need any additional purchases for hardware and software.

In this research, Code-blocks IDE has been used. For hardware, the configuration has to be minimum RAM 4 GB, hard disk space 1.5 GB + caches at least 1 GB. Now for the benefit, this system will help the Tokens, Lexemes, Buffering, Regular Definitions.Regular Expressions.Patterns.etc.

3.3 Requirement Analysis

Requirement analysis is significant and essential activity after elicitation. We analyze, refine, and scrutinize the gathered requirements to make consistent and unambiguous requirements. This activity reviews all requirements and may provide a graphical view of the entire system. After the completion of the analysis, it is expected that the understandability of the project may improve significantly. Here, we may also use the interaction with the customer to clarify points of confusion and to understand which requirements are more important than others.

3.4 Summary

In this section we tried to discuss about all the feasibility analysis like technical feasibility ,economic feasibility, operational feasibility and also requirement analysis. We discussed about data collection, data prepossessing, data understanding and feature selection.

4 Implementation and Testing

4.1 Introduction

In this section we will give an over all idea of system and performance of our proposed model for user input expression.

4.2 System Setup

First we download codeblock and setup this software in our operating system, then we setup also a codeblocks IDE file and extension also. Then we open a project and paste this code and also save user input in file and define this file type in program, then start program and check our user input.

```
int main()
{
    int number1 , number2 , sum , number3 , number4 , exponent_sum
    printf(" Input two number from keyboard " );
    scanf(" %d %d ", & number1 , & number2 );
    sum = number1 + number2 ;
    printf(" sum of two number is : %d ", sum );
    number3 = 3.225687E^-25.25 ;
    number4 = 35687.3256E^-78954 ;
    exponent_sum = number3 + number4 ;
    printf(" summation of two exponential number is : %lf ", exponent_sum );
    return 0 ;
}
```

Figure 4.5: User Input File

4.3 Evaluation

In this segment, we identifying the digit, Tokens , Lexernes, Buffering, Regular Definitions.Regular Expressions.Patterns. then our compiler scanning and read all expression then compiler match this.

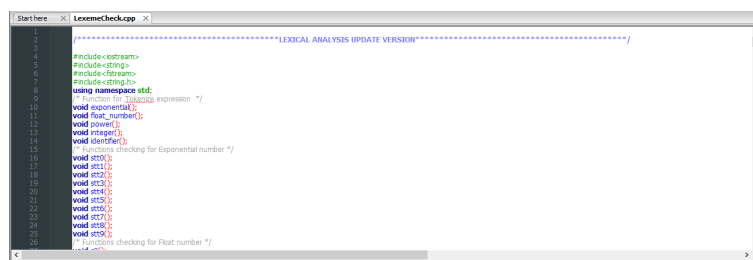
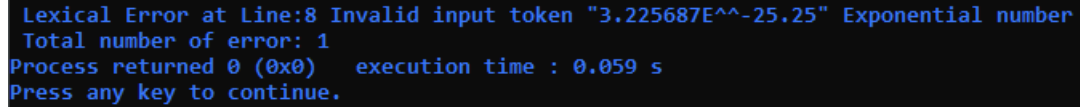


Figure 4.6: Compiler Code Codeblocks Editor

Figure 4.6 showed the output of User Input File.

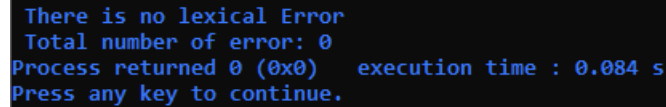
4.4 Results and Discussion

In this result section lexical analyzer detect the user input and compiling this expression. the compiler most of Thousand of line read.

A terminal window with a black background and blue text. The text reads: "Lexical Error at Line:8 Invalid input token "3.225687E^^-25.25" Exponential number", "Total number of error: 1", "Process returned 0 (0x0) execution time : 0.059 s", and "Press any key to continue.".

```
Lexical Error at Line:8 Invalid input token "3.225687E^^-25.25" Exponential number
Total number of error: 1
Process returned 0 (0x0) execution time : 0.059 s
Press any key to continue.
```

Figure 4.7: Error Output From Codeblocks Compiler

A terminal window with a black background and blue text. The text reads: "There is no lexical Error", "Total number of error: 0", "Process returned 0 (0x0) execution time : 0.084 s", and "Press any key to continue.".

```
There is no lexical Error
Total number of error: 0
Process returned 0 (0x0) execution time : 0.084 s
Press any key to continue.
```

Figure 4.8: Without-Error Output From Codeblocks Compiler

4.5 Summary

For the training data, we reach an accuracy of 99.78 percent, and for the test data, it is 99.50 percent. The results are unquestionably good, yet they fall short of what we previously attained. Convolutional neural networks are adaptations of neural networks that use the structure of an image as input and make use of numerous hidden layers, some of which are only employed to produce feature maps utilizing the local structure of an image. For the training data, a simple convolutional neural network produces accuracy of 99.40 percent.

5 Conclusion

5.1 Conclusion

A lexical analyzer is a checker that analyze the code written in one programming language to another. Ex, g++ from GNU family of compilers, Power BASIC, etc. There are 6 phases in the compiler, namely, lexical analysis, syntax analysis, semantics analysis, intermediate code generation, code optimization, and code generation. Symbol table is a data structure that the compiler generates and maintains to keep track of the semantics of variables.

The error Handling routine detects errors, reports them to the user, and follows some recovery plan to handle the errors. Errors could occur either during compile-time or run-time.

The First method Lexical analyzer checking user inputs and check also coding.

There are four methods of recovering from an error- panic mode recovery, phase level recovery, error production, and global correction.

5.2 Limitation and Future Works

5.2.1 Limitation

The issue is that there's a wide range of user input – good and bad. This makes it tricky for programmers to provide enough examples of how every character might look. Plus, sometimes, characters look very similar, making it hard for a computer to recognise accurately.

5.2.2 Future Works

Recently user input expression digit wide range becomes vital scope and it is appealing many researchers because of its using in variety of machine learning and computer vision applications. However, there are deficient works accomplished on Arabic pattern digits because Arabic digits are more challenging than English patterns.

References

[1]. Aho, A. V., S. C. Johnson, and J. D. Ullman, "Deterministic parsing of ambiguous grammars," *Comm. ACM* 18:8 (Aug., 1975), pp. 441-452.

2. Backus, J.W., "The syntax and semantics of the proposed international algebraic language of the Zurich-ACM-GAMM Conference," *Proc. Intl. Conf. Information Processing*, UNESCO, Paris, (1959) pp. 125-132.

3. Birman, A. and J. D. Ullman, "Parsing algorithms with backtrack," *Information and Control* 23:1 (1973), pp. 1-34.

4. Cantor, D. C., "On the ambiguity problem of Backus systems," *J. ACM* 9:4 (1962), pp. 477-479.

5. Chomsky, N., "Three models for the description of language," *IRE Trans. on Information Theory* IT-2:3 (1956), pp. 113-124.

6. Choudhary, A.; Ahlawat, S.; Rishi, R. A binarization feature extraction approach to OCR: MLP vs. RBF. In *Proceedings of the International Conference on Distributed Computing and Technology ICDCIT*, Bhubaneswar, India, 6–9 February 2014; Springer: Cham, Switzerland, 2014; pp. 341–346.

7. Dain, J., "Bibliography on Syntax Error Handling in Language Translation Systems," 1991. Available from the *comp.compilersnewsgroup*; see <http://compilers.iecc.com/comparch/O4-O5O>.

8. Jarrett, K.; Kavukcuoglu, K.; Ranzato, M.; LeCun, Y. What is the best multi-stage architecture for object recognition. In *Proceedings of the IEEE 12th International Conference on Computer Vision (ICCV)*, Kyoto, Japan, 29 September– 2

October 2009.

9. Ciresan, D.C.; Meier, U.; Masci, J.; Gambardella, L.M.; Schmidhuber, J. High-performance neural networks for visual object classification. arXiv 2011, arXiv:1102.0183v1.
10. Ciresan, D.C.; Meier, U.; Schmidhuber, J. Multi-column deep neural networks for image classification. arXiv 2012, arXiv:1202.2745.
11. Niu, X.X.; Suen, C.Y. A novel hybrid CNN–SVM classifier for recognizing handwritten digits. *Pattern Recognit.* 2012, 45, 1318–1325. [CrossRef]
12. Qu, X.; Wang, W.; Lu, K.; Zhou, J. Data augmentation and directional feature maps extraction for in-air handwritten Chinese character recognition based on convolutional neural network. *Pattern Recognit. Lett.* 2018, 111, 9–15. [CrossRef]
13. Alvear-Sandoval, R.; Figueiras-Vidal, A. On building ensembles of stacked denoising auto-encoding classifiers and their further improvement. *Inf. Fusion* 2018, 39, 41–52. [CrossRef]
14. Demir, C.; Alpaydin, E. Cost-conscious classifier ensembles. *Pattern Recognit. Lett.* 2005, 26, 2206–2214. [CrossRef]
15. Choudhary, A.; Ahlawat, S.; Rishi, R. A neural approach to cursive handwritten character recognition using features extracted from binarization technique. *Complex Syst. Model. Control Intell. Soft Comput.* 2015, 319, 745–771.
16. Choudhary, A.; Rishi, R.; Ahlawat, S. Handwritten numeral recognition using modified BP ANN structure. In *Proceedings of the Communication in Computer and Information Sciences (CCIS-133), Advanced Computing, CCSIT 2011*, Royal Orchid Central, Bangalore, India, 2–4 January 2011; Springer: Berlin/Heidelberg,

Germany, 2011; pp. 56–65.

17. Cai, Z.W.; Li-Hong, H. Finite-time synchronization by switching state-feedback control for discontinuous Cohen–Grossberg neural networks with mixed delays. *Int. J. Mach. Learn. Cybern.* 2018, 9, 1683–1695. [CrossRef]

18. Zeng, D.; Dai, Y.; Li, F.; Sherratt, R.S.; Wang, J. Adversarial learning for distant supervised relation extraction. *Comput. Mater. Contin.* 2018, 55, 121–136.

19. Long, M.; Yan, Z. Detecting iris liveness with batch normalized convolutional neural network. *Comput. Mater. Contin.* 2019, 58, 493–504. [CrossRef]

20. Chuangxia, H.; Liu, B. New studies on dynamic analysis of inertial neural networks involving non-reduced order method. *Neurocomputing* 2019, 325, 283–287.

21. Xiang, L.; Li, Y.; Hao, W.; Yang, P.; Shen, X. Reversible natural language watermarking using synonym substitution and arithmetic coding. *Comput. Mater. Contin.* 2018, 55, 541–559.

22. Schorre, D. V., "Meta-11: a syntax-oriented compiler writing language," *Proc. 19th ACM Natl. Conf.* (1964) pp. D1.3-1-D1.3-11.

23. Choudhary, A.; Rishi, R. Improving the character recognition efficiency of feed forward bp neural network. *Int. J. Comput. Sci. Inf. Technol.* 2011, 3, 85–96. [CrossRef]

24. Ahlawat, S.; Rishi, R. A genetic algorithm based feature selection for handwritten digit recognition. *Recent Pat. Comput. Sci.* 2019, 12, 304–316. [CrossRef]

25. Hinton, G.E.; Osindero, S.; Teh, Y.W. A fast learning algorithm for deep belief

nets. *Neural Comput.* 2006, 18, 1527– 1554. [CrossRef]

26. Pham, V.; Bluche, T.; Kermorvant, C.; Louradour, J. Dropout improves recurrent neural networks for handwriting recognition. In *Proceedings of the 14th Int. Conf. on Frontiers in Handwriting Recognition*, Heraklion, Greece, 1–4 September 2014.

27. Tabik, S.; Alvear-Sandoval, R.F.; Ruiz, M.M.; Sancho-Gómez, J.L.; Figueiras-Vidal, A.R.; Herrera, F. MNISTNET10: A heterogeneous deep networks fusion based on the degree of certainty to reach 0.1 Overv. *Proposal Inf. Fusion* 2020, 62, 73–80. [CrossRef]

28. Lang, G.; Qingguo, L.; Mingjie, C.; Tian, Y.; Qimei, X. Incremental approaches to knowledge reduction based on characteristic matrices. *Int. J. Mach. Learn. Cybern.* 2017, 8, 203–222. [CrossRef]

29. Badrinarayanan, V.; Kendall, A.; Cipolla, R. SegNet: A Deep convolutional encoder-decoder architecture for image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.* 2017, 39, 2481–2495. [CrossRef]

30. He, S.; Zeng, W.; Xie, K.; Yang, H.; Lai, M.; Su, X. PPNC: Privacy preserving scheme for random linear network coding in smart grid. *KSII Trans. Internet Inf. Syst.* 2017, 11, 1510–1532.

31. Sueiras, J.; Ruiz, V.; Sanchez, A.; Velez, J.F. Offline continuous handwriting recognition using sequence to sequence neural networks. *Neurocomputing.* 2018, 289, 119–128. [CrossRef]

32. Liang, T.; Xu, X.; Xiao, P. A new image classification method based on modified condensed nearest neighbor and convolutional neural networks. *Pattern Recognit.*

33. Simard, P.Y.; Steinkraus, D.; Platt, J.C. Best practice for convolutional neural networks applied to visual document analysis. In Proceedings of the Seventh International Conference on Document Analysis and Recognition (ICDAR 2003), Edinburgh, UK, 3–6 August 2003.
34. Wang, T.; Wu, D.J.; Coates, A.; Ng, A.Y. End-to-end text recognition with convolutional neural networks. In Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012), Tsukuba, Japan, 11–15 November 2012.
35. Shi, B.; Bai, X.; Yao, C. An End-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. *IEEE Trans. Pattern Anal. Mach. Intell.* 2017, 39, 2298–2304. [CrossRef]
36. Long, J.; Shelhamer, E.; Darrell, T. Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015.
37. Chen, L.-C.; Papandreou, G.; Kokkinos, I.; Murphy, K.; Yuille, A.L. DeepLab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs. *IEEE Trans. Pattern Anal. Mach. Intell.* 2018, 40, 834–848. [CrossRef]
38. Noh, H.; Hong, S.; Han, B. Learning deconvolution network for semantic segmentation. In Proceedings of the IEEE International Conference on Computer Vision, Araucano Park, Las Condes, Chile, 11–18 December 2015.
39. Boufenar, C.; Kerboua, A.; Batouche, M. Investigation on deep learning for off-line handwritten Arabic character recognition. *Cogn. Syst. Res.* 2018, 50,

180–195. [CrossRef]

40. Kavitha, B.; Srimathi, C. Benchmarking on offline Handwritten Tamil Character Recognition using convolutional neural networks. *J. King Saud Univ. Comput. Inf. Sci.* 2019. [CrossRef]

41. Dewan, S.; Chakravarthy, S. A system for offline character recognition using auto-encoder networks. In *Proceedings of the International Conference on Neural Information Processing*, Doha, Qatar, 12–15 November 2012.

42. Ahmed, S.; Naz, S.; Swati, S.; Razzak, M.I. Handwritten Urdu character recognition using one-dimensional BLSTM classifier. *Neural Comput. Appl.* 2019, 31, 1143–1151. [CrossRef]

43. Husnain, M.; Saad Missen, M.; Mumtaz, S.; Jhanidr, M.Z.; Coustaty, M.; Luqman, M.M.; Ogier, J.-M.; Choi, G.S. Recognition of urdu handwritten characters using convolutional neural network. *Appl. Sci.* 2019, 9, 2758. [CrossRef]

44. Sarkhel, R.; Das, N.; Das, A.; Kundu, M.; Nasipuri, M. A multi-scale deep quad tree based feature extraction method for the recognition of isolated handwritten characters of popular indic scripts. *Pattern Recognit.* 2017, 71, 78–93. [CrossRef]

45. Xie, Z.; Sun, Z.; Jin, L.; Feng, Z.; Zhang, S. Fully convolutional recurrent network for handwritten chinese text recognition. In *Proceedings of the 23rd International Conference on Pattern Recognition (ICPR 2016)*, Cancun, Mexico, 4–8 December 2016.

46. Liu, C.; Yin, F.; Wang, D.; Wang, Q.-F. Online and offline handwritten Chinese character recognition: Benchmarking on new databases. *Pattern Recognit.* 2013, 46, 155–162. [CrossRef]

47. Wu, Y.-C.; Yin, F.; Liu, C.-L. Improving handwritten chinese text recognition using neural network language models and convolutional neural network shape models. *Pattern Recognit.* 2017, 65, 251–264. [CrossRef]
48. Gupta, A.; Sarkhel, R.; Das, N.; Kundu, M. Multiobjective optimization for recognition of isolated handwritten Indic scripts. *Pattern Recognit. Lett.* 2019, 128, 318–325. [CrossRef]
49. Nguyen, C.T.; Khuong, V.T.M.; Nguyen, H.T.; Nakagawa, M. CNN based spatial classification features for clustering offline handwritten mathematical expressions. *Pattern Recognit. Lett.* 2019. [CrossRef]
50. Ziran, Z.; Pic, X.; Innocenti, S.U.; Mugnai, D.; Marinai, S. Text alignment in early printed books combining deep learning and dynamic programming. *Pattern Recognit. Lett.* 2020, 133, 109–115. [CrossRef]
51. Ptucha, R.; Such, F.; Pillai, S.; Brokler, F.; Singh, V.; Hutkowski, P. Intelligent character recognition using fully convolutional neural networks. *Pattern Recognit.* 2019, 88, 604–613. [CrossRef]
52. Cui, H.; Bai, J. A new hyperparameters optimization method for convolutional neural networks. *Pattern Recognit. Lett.* 2019, 125, 828–834. [CrossRef]
53. Tso, W.W.; Burnak, B.; Pistikopoulos, E.N. HY-POP: Hyperparameter optimization of machine learning models through parametric programming. *Comput. Chem. Eng.* 2020, 139, 106902. [CrossRef]
54. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern*

Recognition (CVPR), Las Vegas, NV, USA, 26 June–1 July 2016.

55. Christian, S.; Wei, L.; Yangqing, J.; Pierre, S.; Scott, R.; Dragomir, A.; Andrew, R. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015.

56. Simonyan, K.; Zisserman, A. Very deep convolutional networks for large-scale image recognition. In Proceedings of the International Conference on Learning Representations, San Diego, CA, USA, 7–9 May 2015.

57. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. ImageNet classification with deep convolutional neural networks. *Adv. Neural Inf. Process. Syst.* 2012, 25, 1097–1105. [CrossRef]

58. Eickenberg, M.; Gramfort, A.; Varoquaux, G.; Thirion, B. Seeing it all: Convolutional network layers map the function of the human visual system. *NeuroImage* 2017, 152, 184–194. [CrossRef] [PubMed]15 of 15

59. Le, H.; Borji, A. What are the receptive, effective receptive, and projective fields of neurons in convolutional neural networks? *arXiv* 2018, arXiv:1705.07049.

60. Luo, W.; Li, Y.; Urtasun, R.; Zemel, R. Understanding the effective receptive field in deep convolutional neural networks. *arXiv* 2017, arXiv:1701.04128.

61. Lin, G.; Wu, Q.; Qiu, L.; Huang, X. Image super-resolution using a dilated convolutional neural network. *Neurocomputing* 2018, 275, 1219–1230. [CrossRef]

62. Scherer, D.; Muller, A.; Behnke, S. Evaluation of pooling operations in convolutional architectures for object recognition. In Proceedings of the International Conference on Artificial Neural Networks, Thessaloniki, Greece, 15–18 September

2010.

- 63. Shi, Z.; Ye, Y.; Wu, Y. Rank-based pooling for deep convolutional neural networks. *Neural Netw.* 2016, 83, 21–31. [CrossRef]
- 64. Wu, H.; Gu, X. Towards dropout training for convolutional neural networks. *Neural Netw.* 2015, 71, 1–10. [CrossRef] [PubMed]
- 65. Saeed, F.; Paul, A.; Karthigaikumar, P.; Nayyar, A. Convolutional neural network based early fire detection. *Multimed. Tools Appl.* 2019, 1–17. [CrossRef]
- 66. Alzubi, J.; Nayyar, A.; Kumar, A. Machine learning from theory to algorithms: An overview. *J. Phys. Conf. Series* 2018, 1142, 012012. [CrossRef]
- 67. Duchi, J.; Hazen, E.; Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.* 2011, 12, 2121–2159.
- 68. Zeiler, M.D. ADADELTA: An Adaptive Learning Rate Method. *arXiv* 2012, arXiv:abs/1212.5701.
- 69. Kingma, D.; Ba, J. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations*, San Diego, CA, USA, 7–9 May 2015.
- 70. Bartlett, P.; Hazan, E.; Rakhlin, A. Adaptive online gradient descent. In *Proceedings of the NIPS*, Vancouver, BC, Canada, 8–11 December 2008.
- 71. Grune, D and C. J. H. Jacobs, "A programmer-friendly LL(1) parser generator," *Software Practice and Experience* 18:1 (Jan., 1988), pp. 29- 38. See also <http://www.cs.vu.nl/~ceriel/LLgen.html> .

72. Johnson, S. C., "Yacc- Yet Another Compiler Compiler," Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Available at <http://dinosaur.compilertools.net/yacc/> .
73. Shalev-Shwartz, S.; Singer, Y.; Srebro, N. Pegasos: Primal estimated sub-gradient solver for svm. In Proceedings of the ICML, Corvallis, OR, USA, 20–24 June 2007.
74. Zinkevich, M.; Weimer, M.; Smola, A.; Li, L. Parallelized stochastic gradient descent. NIPS 2010, 2, 2595–2603.
75. LeCun, Y.; Boser, B.; Denker, J.S.; Henderson, D. Backpropagation applied to handwritten zip code recognition. Neural Comput. 1989, 1, 541–551. [CrossRef]
76. Dietterich, T.; Bakiri, G. Solving multiclass learning problems via error-correcting output codes. J. Artif. Intell. Res. 1995, 1, 263–286. [CrossRef]
77. Korenjak, A. J., "A practical method for constructing LR(k) processors," Comm. ACM 12:11 (Nov., 1969),pp. 613-623
78. Naur, P. et al., "Report on the algorithmic language ALGOL 60," Comm. ACM 3:5 (May, 1960), pp. 299-314. See also Comm. ACM 6:1 (Jan., 1963), pp. 1-17.
79. Wirth, N. and H. Weber, "Euler: a generalization of Algol and its formal definition: Part I," Comm. ACM 9:1 (Jan., 1966),pp. 13-23