Code Clone Detection with Convolutional Neural Networks

Araştırma Makalesi/Research Article

Harun DİŞLİ, Ayşe TOSUN

Faculty of Informatics and Computer Engineering, Istanbul Technical University, Istanbul, Turkey
disli15@itu.edu.tr, tosunay@itu.edu.tr
(Geliş/Received:18.03.2019; Kabul/Accepted:25.10.2019)
DOI: 10.17671/gazibtd.541476

***Abstract*—** Similar or identical code portions which are generated by copying and reusing code portions within the source code are named as code clones. While so many works have been conducted to detect these clones, they generally use string comparison techniques and very few of them take advantage of popular learning based approaches, such as deep learning. This paper proposes a new approach based on a popular and successful image classification technique named as convolutional neural network. It simply tokenizes each candidate clone pair in order to generate image files. Then, convolutional neural network is used to classify these image data with labels "clone" and "not clone". In order to train and test the network, clone and not clone pairs are chosen from a public database including six million methods. As a result, the approach gives 99% accuracy, effectively detects clones and not clones with 2-5% false alarms rates at method granularity.

Keywords— code clone detection, deep learning, convolutional neural network

Konvolüsyonel Sinir Ağları İle Kod Klonlarının Tespiti

***Özet*—** Yazılım geliştirirken kopyalama ve yeniden kullanma yoluyla oluşturulan benzer veya aynı kod parçaları, kod klonları olarak adlandırılır. Bu klonları tespit etmek için pek çok çalışma yapılmış olsa da, çalışmalar genellikle katar karşılaştırma tekniklerini kullanılmakta ve çok azı popüler araştırma alanlarından olan derin öğrenmeden faydalanmaktadır. Bu makale, konvolüsyonel sinir ağı olarak adlandırılan, popüler ve başarılı görüntü sınıflandırma yöntemine dayanan yeni bir yaklaşım önermektedir. Bu yöntem, görüntü dosyalarını oluşturmak için her aday klon çiftini sembollere ayırır. Daha sonra, konvolüsyonel sinir ağı bu görüntü verilerini "klon" veya "klon değil" etiketleriyle sınıflandırmak için kullanılır. Ağı eğitmek ve test etmek için altı milyon java metodu içeren bir veri tabanından örneklerler seçilerek kullanılmıştır. Sonuç olarak, bu yaklaşım metot bazındaki klonları % 95'lik bir doğrulukla etkili bir şekilde tespit etmektedir.

***Anahtar Kelimeler*—** kod klon tespiti, derin öğrenme, konvolüsyonel sinir ağı

1. INTRODUCTION

Copying and reusing a code fragment with or without minor modification is known as code cloning [1]. Code clones generally occur as a result of copy-paste operations by programmers [2]. When a code piece has residual bugs, it is highly possible that the same problem also exists in the other copies [2]. Accordingly, when an update is required in a code piece which has multiple copies in the source code, the developer should check the other copies to avoid unexpected crashes in the system. There are four different clone types, which are Type-1, Type-2, Type-3 and Type-4. Type-1 clones can be identical or they may contain little modification on whitespace and comments [3]. Type-2 clones allow changing variables, type or identifiers. Type-3 clones extend this with adding, removing or changing some statements [3,4]. Type-4 clone perform same operation with different syntactic variants [4]. In a software system that contains high number of clones and developed by a large group of programmers, the maintenance cost may be seriously high because of these duplicated code portions [5]. Detecting code clones in a software system is, therefore, crucial to reduce both significant maintenance

costs and the risks of potential failures associated with the clone operations.

There have been several studies on code clone detection employing different approaches to identify cloned code pieces in large scale software applications. Some of the approaches proposed in the studies perform text-based techniques [6-8], while others use token-based techniques [9-12], semantic or tree-based techniques [13,14]. Text-based techniques treat source code as strings consisting of words and compare strings among each other to identify their similarities [15]. The other approaches transform the source code into tokens, or graphical models before applying an algorithm to detect cloned pieces. To build the code clone detection model, algorithms such as suffix tree [10,16], dotplot/scatter plot [17], hash-value comparison [13,14] and Euclidean distance [18] have been tried. Recently, studies show that deep learning techniques such as RNN (recurrent neural network) [19,20] and LSTM (long short-term memory network) [21] are used for modeling source codes. One particular field that requires modeling the source code is code clone detection. Studies on clone detection using deep learning techniques, such as DNN (deep neural network) [22] report that at least 98% of Type-1 and Type-2 clones can be predicted.

It was stated that there are mostly Type-3 clones in source code [23]. However, this clone type has the lowest prediction performances with Type-4 clone type: Even with deep learning, CClearner [22] has reached a maximum detection rate of 28% in weakly Type-3 and Type-4 clones. For that reason, further investigation is necessary in code clone detection, specifically for detecting all types of code clones with high accuracy.

This paper proposes a token-based technique using convolutional neural network (CNN) on code clone detection by converting source code into the form of image data. To implement this approach, we used candidate clone pair methods extracted from BigCloneBench dataset [25], and tokenized these methods. Number of occurrences of every token counted for both methods separately and saved as frequency matrices. Later, these two matrices merged and saved as an image data. Lastly, CNN is trained using these images that represent clone pairs. Since the merged matrix appears very similar to image data, we use it as if it is an image. Also, we preferred CNN because it assumes the inputs are images and it provides encoding features to the architecture. This helps reduce the number of parameters and make more efficient implementation [26]. Empirical analysis on three different train-test splits used with CNN shows that our approach has good performance on detecting code clones especially complicated clone types.

The rest of the paper is organized as follow. Section 2 presents brief background information about code clone detection and related works. Section 3 expresses the proposed clone detection technique. Section 4 and 5 show our experimental results and comparison with similar works. Lastly, section 6 summarizes and concludes this paper.

2. BACKGROUND INFORMATION

2.1. Code Clone Definitions

Code clone definitions and clone types are explained as follows:

Code Fragment: Any grouping of code lines in any granularity such as function definition, begin_end block, or sequence of statements named as code fragment(CF). A CF is recognized by its file name and begin_end line numbers in the original code base. It is signified as a triple (CF.FileName, CF.BeginLine, CF.EndLine) [4]. In this work, the fragments are equal to methods (functions).

Code Clone: If two fragments are similar or identical they are called as code clone. In another words, if two fragments are code clone, it should be that f(CF1) = f(CF2) where f is the similarity function. Code clone may be among two or more code fragments. While a clone pair is formed by two similar code fragments (CF1, CF2), clone group or clone class is formed by many similar fragments [4].

Clone Types: Clone types between fragments can be grouped as two main kinds which are based on the similarity of their program text and similarity of their functionality (independent of their text). [4]. The textual (Types 1 to 3) and functional (Type-4) clone types are provided in the following:

Type-1: Two fragments are Type-1 clone if they are identical except for variations in whitespace, layout and comments [4]. In Figure 1.a, whitespace is the only difference between two methods.

Type-2: If two fragments are varied with only identifiers, literals, types, whitespace, layout and comments, they are Type-2 clones [4]. Figure 1.b is an example of Type-2 clone. Both methods copy given files in same way, but variable and parameter names are different. Regarding our results and previous researches [3,4], Type-1 and Type-2 clones are simplest types to detect.

Type-3: If a fragment can be obtained from another with little modifications such as changed, added or removed statements, these two fragments are Type-3 clone [4]. Figure 1.c shows an example of Type-3 clone. Added line 3 and modified line 4 in second method are variations between two methods.

Type-4: Two or more code fragments may perform the same computation by implementing different syntactic variants. These types of fragments are known as Type-4 clone [4]. In Figure 1.d, both methods perform same operation in different ways.

```java
public String stringOfUrl(String addr) throws IOException {
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    URL url = new URL(addr);
    IOUtils.copy(url.openStream(), output);
    return output.toString();
}
```

```java
public String stringOfUrl(String addr) throws IOException {
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    URL url = new URL(addr);
    IOUtils.copy(url.openStream(), output);

    return output.toString();
}
```

(a)

```java
public static void copyFile(String src, String target) throws IOException {
    FileChannel ic = new FileInputStream(src).getChannel();
    FileChannel oc = new FileOutputStream(target).getChannel();
    ic.transferTo(0, ic.size(), oc);
    ic.close();
    oc.close();
}
```

```java
public void copyFile(File in, File out) throws Exception {
    FileChannel sourceChannel = new FileInputStream(in).getChannel();
    FileChannel destinationChannel = new ileOutputStream(out).getChannel();
    sourceChannel.transferTo(0, sourceChannel.size(), destinationChannel);
    sourceChannel.close();
    destinationChannel.close();
}
```

(b)

```java
public String streamToString(InputStream stream) throws IOException {
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    IOUtils.copy(stream, output);
    return output.toString();
}
```

```java
public String stringOfUrl(String addr) throws IOException {
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    URL url = new URL(addr);
    IOUtils.copy(url.openStream(), output);
    return output.toString();
}
```

(c)

```java
public static final String getMD5Hash(byte[] data, int offset, int length) throws NoSuchAlgorithmException {
    MessageDigest md5 = MessageDigest.getInstance("MD5");
    md5.update(data);
    return generateHash(md5.digest());
}

public String getHash(String key, boolean base64) throws Exception {
    MessageDigest md = MessageDigest.getInstance("SHA");
    md.update(key.getBytes());
    if (base64) return new String(new Base64().encode(md.digest()), "UTF8");
    else return new String(md.digest(), "UTF8");
}
```

(d)

Figure 1. Clone Types. Type-1 clones (a), Type-2 clones (b), Type-3 clones (c) and Type-4 clones (d),

2.2. Related Work on Code Clone Detection

Code clone detection approaches may be separated into five main categories regarding the analysis to transform source code: textual, lexical, syntactic, semantic and hybrid [4] as shown in Figure 2.

Textual approaches take each line and use string matching algorithms. These techniques use raw source code and apply little or no transformation to source code. While they manage to detect Type-1 clone, they hardly detect complicated types [4,22]. Ducasse et al. [15] use text based clone detection approach. The detection is based on construction of dot plots whose axes are source code entities. If hash values of the entities on two axes are equal, then there is a dot at corresponding coordinate.

Lexical approaches are also known as token-based approaches. They remove white spaces and comments, transform source code to a sequence of tokens using lexical analysis techniques and then search for duplicated subsequences. After locating duplicate subsequences, corresponding original source code portions returned as code clone. One of the efficient token based approaches is CCFinder [10]. It removes all white spaces between tokens, transforms tokens based on some transformation rules. Then, it searches for same transformed token sequences. Detected sequences are marked as duplicate. Another lexical approach is Dup. by Baker [9] that utilizes p-string and p-matching. After removing white spaces and comments, identifiers such as variable, method and class are renamed. Then, each line of codes is hashed for comparison. Using suffix-tree algorithm, duplicated code portions are located. These techniques are more successful detecting complicated clone types [4].

Syntactic approaches can be identified with two sub categories. Tree matching approaches (tree-based techniques) works creating an AST tree for each code fragment and finding similar subtrees. Variable names, literal values and other tokens may be abstracted in tree to find more complicated clone types. Since tree matching approaches focus the syntactic structure of the source code rather than their statements, they can detect near-miss clones [4,22]. CloneDr by Baxter et al [13] is an example for tree matching approaches. First, it transforms the source code into parse tree. Later, subtrees are hashed into buckets. These hashed subtrees are compared in order to identify clone portions.

Metric-based approaches collect a number of metrics for each code fragments. These metrics may be number of statements, number of function calls, number of input variable. Rather than comparing code, this technique

compares metric vectors to identify code clones [4,22]. Davey et al. [27] generates futures for code blocks using metrics and train a neural network to determine duplicated code blocks.

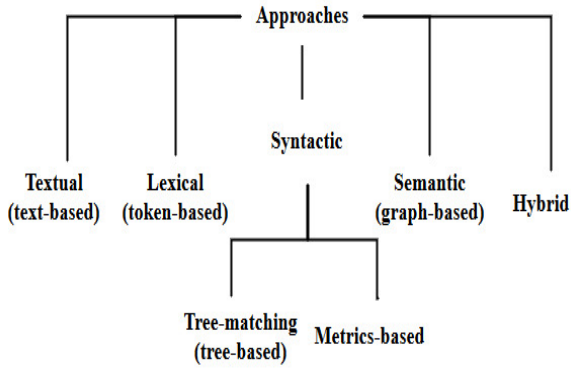

Figure 2. Code Clone Detection Approaches

Semantic based approaches using semantic program analysis represent the software as Program Dependency Graph or Control Flow Graph. Expression and statements transformed as nodes while control and data dependencies transformed as edge in the graph. After this transformation, code clone detection is turned identifying isomorphic subgraphs [4]. Komondoor and Horwitz's tool [28] detects clones using Program Dependency Graph. Hybrid approaches use combination of other four techniques to locate the code clones.

There are also similar works that implemented deep learning. These works may benefit from any clone detection approach. White et al. [29] introduced learning-based detection technique which relies on language model that is a probability distribution over sentences in a language. Recurrent neural network and recursive neural network are used to map each term in a fragment to an embedding and to encode this embedding to characterize fragments.

Another tool that uses deep learning is CCLearner [22]. CCLearner divides tokens into eight categories such as reserved words, operators and markers. It calculates similarity score for each category and obtains frequency vectors with length eight. Obtained vectors are used to train deep neural network. It performs analysis on 1626544 samples taken from dataset BigCloneBench and took 100%, 98%, 98% and 89% recalls for T1, T2, VST3 and ST3 respectively. However, CCLearner performs 28% recall for MT3 and 1% for WST3/4 because it does not include MT3 and WST3/4 clone types in their training set. In addition, CCLearner calculates the overall precision based on 385 randomly selected samples and didn't report false positive rate of their approach. Since Type-3 clone is the most common clone type, we want to develop a method that detects all clone types separately. We applied deep learning like CCLearner, but specifically we thought CNN would be the best fit. We also reported false positive rate in addition to recall and precision. We evaluated our work using the same dataset but with different number of samples. In addition, we utilized tokens frequencies as feature vector without categorization.

3. PROPOSED APPROACH

In this section, we defined BigCloneBench benchmark [25] and CNN. Also, we described our clone detection technique including data preprocessing, how we constructed CNN architecture with its initialization parameters and training and test set used to evaluate our proposed technique.

3.1. Dataset

BigCloneBench is a collection of over six million validated clones in the large Java inter-project repository IJaDataset-2.0. BigCloneBench was built by mining IJaDataset for clones of particular functionalities. It contains both inter-project and intra-project clones of the four primary clone types [25]. The dataset contains almost 2.5 million Java classes with more than 22 million methods. Using these classes and their methods, Svajlenko et al. [25] created comparison tables which contain clone and not clone samples with their types.

In BigCloneBench, there are six extended clone types: T1(Type-1), T2(Type-2), VST3(Very Strong Type-3), ST3(Strong Type-3), MT3(Moderately Type-3) and WT3/4(Weak Type 3 or 4). Svajlenko et al. [25] actually reclassified Type-3 and Type-4 clones based on their similarity ratio. If similarity ratio between two clone candidates more than 0,9, they are labeled as WST3. If the similarity ratio between 0,7 and 0,9, they are ST3. When the ratio is lower than 0,7 and higher than 0,5, they become MT3. If the similarity is lower than 0,5, they are WST3/4. Table 1 shows the distribution of clones based on their types. While there are more than six millions of WT3/4, there are only 2083 VST3 clones. In addition to the clones, Table 1 shows the number of false clone pairs (method pairs that are not clones of each other).

Table 1. Number of clones

Type	Similarity(s) Rule	Number	%
T1	-	16185	0,24
T2	-	3787	0,06
VST3	0,9 <= s <= 1	2083	0,03
ST3	0,7 <= s < 0,9	10031	0,15
MT3	0,5 <= s < 0,7	55106	0,85
WT3/4	0 <= s < 0,5	6158975	94,63
FALSE CLONE	-	262465	4,03

3.2. Tokenization and Input Features

A method pair to be compared is selected and converted to a sequence of tokens, similar to an image data in order to obtain sensible input for CNN. To achieve this, each method pair as method1 and method2 is transformed into tokens with a lexical analyzer tool called ANTLR [30]. Then each token is represented as a unique ID offered by

the tool. Table 2 shows the list of tokens used in a source method, and their corresponding token numbers.

Two sample methods and their tranformation steps are illustrated in Figure 3. First, these two methods are seperated into their tokens. (c) and (d) are token sequences of the first and second methods, respectively. Later, these tokens are replaced with their corresponding unique IDs as shown in (e) and (f). Then token ID sequences obtained for the two methods are merged into a single vector. During the tokenization, each method is illustrated as a token array of size 66 such that the first 66 tokens of a method is represented. We set this number by analysing the average, minimum and maximum number of tokens each method contains. We have found that there are on average 66 tokens in a single method in the BigCloneBench dataset. The final feature set of a method pair, hence, corresponds to a token array of size 132. This token array is later saved as an image in jpeg format.

Table 2. Tokens and corresponding unique IDs

ABSTRACT	1	INTERFACE	28	StringLiteral	55	SUB	82
ASSERT	2	LONG	29	NullLiteral	56	MUL	83
BOOLEAN	3	NATIVE	30	LPAREN	57	DIV	84
BREAK	4	NEW	31	RPAREN	58	BITAND	85
BYTE	5	PACKAGE	32	LBRACE	59	BITOR	86
CASE	6	PRIVATE	33	RBRACE	60	CARET	87
CATCH	7	PROTECTED	34	LBRACK	61	MOD	88
CHAR	8	PUBLIC	35	RBRACK	62	ARROW	89
CLASS	9	RETURN	36	SEMI	63	COLONCOLON	90
CONST	10	SHORT	37	COMMA	64	ADD_ASSIGN	91
CONTINUE	11	STATIC	38	DOT	65	SUB_ASSIGN	92
DEFAULT	12	STRICTFP	39	ASSIGN	66	MUL_ASSIGN	93
DO	13	SUPER	40	GT	67	DIV_ASSIGN	94
DOUBLE	14	SWITCH	41	LT	68	AND_ASSIGN	95
ELSE	15	SYNCHRONIZED	42	BANG	69	OR_ASSIGN	96
ENUM	16	THIS	43	TILDE	70	XOR_ASSIGN	97
EXTEBDS	17	THROW	44	QUESTION	71	MOD_ASSIGN	98
FINAL	18	THROWS	45	COLON	72	LSHIFT_ASSIGN	99
FINALLY	19	TRANSIENT	46	EQUAL	73	RSHIFT_ASSIGN	100
FLOAT	20	TRY	47	LE	74	URSHIFT_ASSIGN	101
FOR	21	VOID	48	GE	75	Identifier	102
IF	22	VOLATILE	49	NOTEQUAL	76	AT	103
GOTO	23	WHILE	50	AND	77	ELLIPSIS	104
IMPLEMENTS	24	IntegerLiteral	51	OR	78	WS	105
IMPORT	25	FloatingPointLiteral	52	INC	79	COMMENT	106
INSTANCEOF	26	BooleanLiteral	53	DEC	80	LINE_COMMENT	107
INT	27	CharacterLiteral	54	ADD	81		

```
static void copy(String src, String dest) throws IOException {
    File ifp = new File(src);
    File ofp = new File(dest);
    if (ifp.exists() == false) {
        throw new IOException("file '" + src + "' does not exist");
    }
    FileInputStream fis = new FileInputStream(ifp);
    FileOutputStream fos = new FileOutputStream(ofp);
    byte[] b = new byte[1024];
    int readBytes;
    while ((readBytes = fis.read(b)) > 0) fos.write(b, 0, readBytes);
    fis.close();
    fos.cloes();
}
```

(a)

```
private static final void cloneFile(File origin, File target)
                                    throws IOException {
    FileChannel srcChannel = null;
    FileChannel destChannel = null;
    try {
        srcChannel = new FileInputStream(origin).getChannel();
        destChannel = new FileOutputStream(target).getChannel();
        destChannel.transferFrom(srcChannel, 0, srcChannel.size());
    } finally {
        if (srcChannel != null) srcChannel.close();
        if (destChannel != null) destChannel.close();
    }
}
```

(b)

```
STATIC VOID Identifier LPAREN Identifier Identifier COMMA Identifier Identifier RPAREN THROWS Identifier LBRACE
        Identifier Identifier ASSIGN NEW Identifier LPAREN Identifier RPAREN SEMI
        Identifier Identifier ASSIGN NEW Identifier LPAREN Identifier RPAREN SEMI
        IF LPAREN Identifier DOT Identifier LPAREN RPAREN EQUAL BooleanLiteral RPAREN LBRACE
            THROW NEW Identifier LPAREN StringLiteral ADD Identifier ADD StringLiteral RPAREN SEMI
        RBRACE
        Identifier Identifier ASSIGN NEW Identifier LPAREN Identifier RPAREN SEMI
        Identifier Identifier ASSIGN NEW Identifier LPAREN Identifier RPAREN SEMI
        BYTE LBRACK RBRACK Identifier ASSIGN NEW BYTE LBRACK IntegerLiteral RBRACK SEMI
        INT Identifier SEMI
        WHILE LPAREN LPAREN Identifier ASSIGN Identifier DOT Identifier LPAREN Identifier RPAREN RPAREN GT IntegerLiteral RPAREN
            Identifier DOT Identifier LPAREN Identifier COMMA IntegerLiteral COMMA Identifier RPAREN SEMI
        Identifier DOT Identifier LPAREN RPAREN SEMI
        Identifier DOT Identifier LPAREN RPAREN SEMI
RBRACE
```

(c)

```
PRIVATE STATIC FINAL VOID Identifier LPAREN Identifier Identifier COMMA Identifier Identifier RPAREN THROWS Identifier LBRACE
        Identifier Identifier ASSIGN NullLiteral SEMI
        Identifier Identifier ASSIGN NullLiteral SEMI
        TRY LBRACE
            Identifier ASSIGN NEW Identifier LPAREN Identifier RPAREN DOT Identifier LPAREN RPAREN SEMI
            Identifier ASSIGN NEW Identifier LPAREN Identifier RPAREN DOT Identifier LPAREN RPAREN SEMI
            Identifier DOT Identifier LPAREN Identifier COMMA IntegerLiteral COMMA Identifier DOT Identifier LPAREN RPAREN
RPAREN SEMI
        RBRACE FINALLY LBRACE
            IF LPAREN Identifier NOTEQUAL NullLiteral RPAREN Identifier DOT Identifier LPAREN RPAREN SEMI
            IF LPAREN Identifier NOTEQUAL NullLiteral RPAREN Identifier DOT Identifier LPAREN RPAREN SEMI
        RBRACE
RBRACE
```

(d)

```
38 48 102 57 102 102 64 102 102 58 45 102 59
        102 102 66 31 102 57 102 58 63
        102 102 66 31 102 57 102 58 63
        22 57 102 65 102 57 58 73 53 58 59
            44 31 102 57 55 81 102 81 55 58 63
        60
        102 102 66 31 102 57 102 58 63
        102 102 66 31 102 57 102 58 63
        5 61 62 102 66 31 5 61 51 62 63
        27 102 63
        50 57 57 102 66 102 65 102 57 102 58 58 67 51 58
            102 65 102 57 102 64 51 64 102 58 63
        102 65 102 57 58 63
        102 65 102 57 58 63
60
```

(e)

```
33 38 18 48 102 57 102 102 64 102 102 58 45 102 59
        102 102 66 56 63
        102 102 66 56 63
        47 59
            102 66 31 102 57 102 58 65 102 57 58 63
            102 66 31 102 57 102 58 65 102 57 58 63
            102 65 102 57 102 64 51 64 102 65 102 57
58 58 63
        60 19 59
            22 57 102 76 56 58 102 65 102 57 58 63
            22 57 102 76 56 58 102 65 102 57 58 63
        60
60
```

(f)

Figure 3. Preprocessing on methods. Two java method copy (a) and cloneFile (b), their tokenized form (c) and (d), unique ID representations according to the list of tokens in Table 2 (e) and (f).

3.3. Training and Test Set Construction

In this work, we tried several methodologies to construct the training data. First, we used real data ratios. Each type has different number of samples in BigCloneBench. For instance, while 94,63% of all data are in the type of WT3/4, only 0,03% of the data are VST3 as shown in Table 1. Keeping the same ratios, we constructed our data with a total of 50000 instances. The first row of Table 3 shows the distributions of instances from different clone types in the training data.

Our second methodology was micro sampling, i.e. undersampling the classes based on the minority class ratio. VST3 has the minimum number of samples which is 2083. Thus, we built our training data choosing 2000 samples for each type, and hence, the total number of instances is 14000 (see Table 4).

Our third methodology takes random samples for each clone type and forms the training data. Regardless of the clone type ratios in the dataset, we randomly picked instances from the main dataset into the training set, and ended up having 25000 method pairs. The Table 5 shows this third methodology's training set.

Table 3. Number of samples for real data ratios

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Train	124	29	16	77	423	47314	47983	2016
Test#1	25	6	3	15	85	9463	9597	403
Test#2	25	6	3	15	85	9463	9597	403
Test#3	25	6	3	15	85	9463	9597	403
Test#4	25	6	3	15	85	9463	9597	403
Test#5	25	6	3	15	85	9463	9597	403
All Test	125	30	15	75	425	47315	47985	2015

Table 4. Number of samples for micro sampling

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Train	2000	2000	2000	2000	2000	2000	12000	2000
Test#1	25	6	3	15	85	9463	9597	403
Test#2	25	6	3	15	85	9463	9597	403
Test#3	25	6	3	15	85	9463	9597	403
Test#4	25	6	3	15	85	9463	9597	403
Test#5	25	6	3	15	85	9463	9597	403
All Test	125	30	15	75	425	47315	47985	2015

Table 5. Number of samples for random ratio

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Train	3063	737	419	1858	6915	12642	25634	24366
Test#1	25	6	3	15	85	9463	9597	403
Test#2	25	6	3	15	85	9463	9597	403
Test#3	25	6	3	15	85	9463	9597	403
Test#4	25	6	3	15	85	9463	9597	403
Test#5	25	6	3	15	85	9463	9597	403
All Test	125	30	15	75	425	47315	47985	2015

To be more realistic, we constructed five different test data for each training based on real ratios. Each test has 25 T1, 6 T2, 3 VST3, 15 ST3, 85 MT3, 9463 WT3/4 and 403 not clone instances. Table 3, Table 4 and Table 5 show the test set distributions in addition to their training sets. The instances in the training set are not included into these test sets to avoid sampling bias.

3.4. Algorithm Selection

In this study, we employed Convolutional Neural Network (CNN) as the algorithm of our code clone detection model. CNN is one of the most popular and successful algorithms for image classification [26, 31, 32]. While it is similar to a typical neural network, the difference between CNN and a neural network is that CNN mostly takes images as input data. This helps encoding certain image properties to architecture and more efficient calculations can be made reducing amount of parameters [26]. Also CNN can analyze images in 3D (width, height and depth). CNN consists of layers applying different operations. Some layers of CNN with their operations are described below.

Alternative deep learning methods are also available, such as Recurrent Neural Networks (RNN), and its variations, i.e., Long Short-Term Memory (LSTM). These networks have been employed in different software engineering problems, such as classification of security vulnerabilities [33] However, such RNN models require time-dependent and sequential data. In our study, the code fragments, i.e., method pairs, do not represent a sequential development: two selected methods are developed independently, and the similarity between them is later predicted via the trained model Therefore, an RNN-style model is not suitable for our study. Deep neural networks could have been an option for our problem, but as mentioned before [26], CNN gives more efficient training cycles with the convolution and pooling layers. Furthermore, CNN preserves spatial relantionship between pixels in an image [34, 35]. This resembles with the representation of a method such that the relationship between the tokens used in a method should be captured by the selected model. We represent a method as a sequence of tokens, and need a model that could capture more informative features through the relationship between the tokens. Hence, we decided that CNN is a suitable fit for the clone detection problem.

3.4.1. Comvolution Layer

Convolution layer is one of the most significant layers that whole architecture called the same name. It can be interpreted as set of filters. Each neuron(filter) takes input images and dot product with weight matrix [26]. When an input image has shape W1*H1*D1 and with K filters (F*F*D1), it produces W2*H2*D2 where

$$W2=(W1-F)/S+1 \text{ (S is stride size)}$$
$$H2=(H1-F)/S+1$$
$$D2=K$$

3.4.2. Pooling Layer

Pooling layer is a down sampling operation. It makes the operation more manageable reducing parameter numbers and helps to control over fitting. It can be max pooling (taking the maximum of pixels in a region) or average pooling (taking mean of pixel in a region) [26]. When an input image has shape W1*H1*D1 and with filters size F*F*D1, it produces W2*H2*D2 where

W2=(W1−F)/S+1 (S is stride size)
H2=(H1−F)/S+1
D2=D1

3.4.3. RELU Layer

It applies elementwise RELU (rectified linear unit) which is an activation function. It leaves data size unchanged [26].

3.4.4. Fully Connected Layer

It is an ordinary neural network that used for classification [26]. It generates scores for each class. If ten classes exist, the output should be 1*1*10.

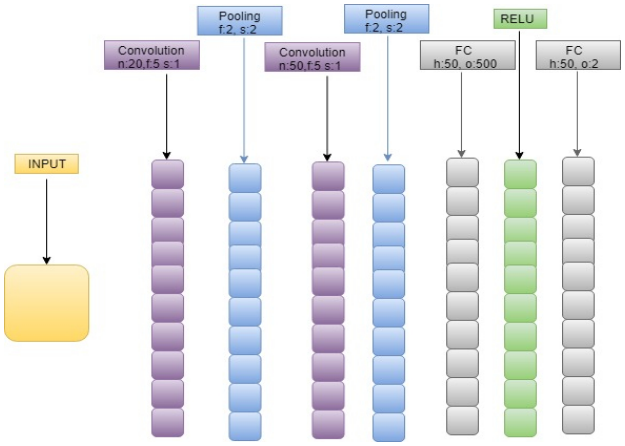

Figure 4. Illustration of CNN architecture

3.5. CNN Architecture

We used Lenet [36] architecture that has two convolutional layers. Every convolution is followed by a pooling layer. The first convolution has 20 filters with size 5*5 and stride 1, second convolution has 50 filters with size 5*5 and stride 1. Each pooling layer is max pooling with kernel size 2*2 and stride 2. After the second pooling layer, we have two fully connected layers. The first one consists of 50 hidden layers and 500 output layers and second one has 50 hidden layers with 2 two outputs. Also, we have a RELU between this two fully connected layers. Lastly, the final layer connected to a softmax to calculate loss. As a shorthand notation C(20,5,1) -> P(2,2) -> C(50,5,1) -> P(2,2) -> FC(50,10) -> R() -> FC(50,2) where C(n,f,s) indicates a convolution layer with n filter, f*f spatial size, s stride; R indicates RELU; P(f,d) indicates pooling with f*f spatial size; s stride and FC(h,o) indicates fully connected layer with h hidden, o output nodes. Figure 4. shows the illustration of the model.

We also tried several different architecture modifying Lenet [36]. When we increased the kernel size in a convolution layer accuracy decreased to 93% and when we reduced the kernel size, accuracy decreased to 91,5%. When we added and removed one convolution layer with pooling, accuracy rates were 91% and 87%, respectively. Additionally, the result of inreasing the stride size by 1 was 92% accuracy.

3.6. Initialization and Optimization of Parameters

We applied parameter tuning for initialization and optimization in the CNN. All of the fully connected and convolution layers' weights are initialized with Xavier initialization [37]. Xavier initilizaiton sets weights of layer to values chosen from random uniform distribution. We also used Adam momentum update [38] with momentum 0,9 and learning rate 0,01. Adam dynamically changes the learning rate and it provides efficient computation with little memory requirements. Batch size was 64.

3.7. Performance Evaluation

We use recall, precision, false positive rate and accuracy as our performance metrics in this work. Since our dataset consists of two labels which are "clone" and "not clone", we defined our metrics for both clone and not clone samples.

Recall (R), measures how many of clones are detected and how many of not clones are detected.

$$R_{clone} = \frac{\text{Number of Clones Truly Predicted}}{\text{Total Number of Clones}} \quad (1)$$

$$R_{notclone} = \frac{\text{Number of Not Clones Truly Predicted}}{\text{Total Number of Not Clones}} \quad (2)$$

Precision (P), measures how many of detected clones are actually clone and how many of detected not clones are actually not clone.

$$P_{clone} = \frac{\text{Number of Clones Truly Predicted}}{\text{Total Number of Predicted Clones}} \quad (3)$$

$$P_{notclone} = \frac{\text{Number of Not Clones Truly Predicted}}{\text{Total Number of Predicted Not Clones}} \quad (4)$$

False Alarm Rate (F), measures how many of the actual not clones are predicted as clone, and how many of the actual clones are predicted as not clone.

$$F_{clone} = \frac{\text{Number of Clones Falsely Predicted}}{\text{Total Number of Not Clones}} \quad (5)$$

$$F_{notclone} = \frac{\text{Number of NotClones Falsely Predicted}}{\text{Total Number of Clones}} \quad (6)$$

Accuracy (A), is an overall measure to compute how many of samples are truly classified.

$$A = \frac{\text{Number of Truly Predicted Samples}}{\text{Total Number of Samples}} \quad (7)$$

4. RESULTS

We performed the experiments on three different training sets as described in Section 3. For each of the experiments, we computed the performance metrics, and reported these in Tables 6-12 except Table 9. Table 9 reports the performance of CCLearner to compare our findings with the recent approach. Tables 6, 7 and 8 report recall rates with respect to clone types, and the overall recall for each experiment. In addition, Tables 10, 11 and 12 show precision, false alarm rate and accuracy for each experiment.

In the experiment using real data ratios, we got 100% recall and 100% precision for the clone samples, as reported in Table 6 and Table 10 respectively. The recall and precision for not clone samples are slightly lower, 96% and 89% respectively. The reason is that we kept the real ratio of clone types in the first methodology, and hence, didn't provide sufficient not clone samples in training data. In the other two trials, we achieved more promising results. We got 99% accuracy for micro sampling and randomly selected samples. Table 7 and Table 8 show recall for each clone type and overall recall. Also, Table 11 and Table 12 show precision, false alarm rate and accuracy for these two experiments. Precision also reached above 80% for not clone samples. This shows that the sampling in the training set has worked well in predicting both clone and not-clone classes successfully. While randomly selected training set has similar recall and precision values for not clone samples, micro sampling reduces the false alarm rates for the clone samples.

The precision and recall values with micro sampling strategy are 83% and 97% on not-clone samples. The recall indicates the truly predicted clone/not-clone samples, and it is more important to predict the class accurately. The precision, on the other hand, indicates the amount of false predictions the model gives. Thus, a tradeoff between the two is important to obtain effective predictions. The change in recall and precision rates on a single test set is illustrated in Figure 5 with varying thresholds. It is seen that, as the model achieves higher recall rates, its precision slightly decreases.

Table 6. Recall for real data ratio (%)

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Test#1	100	100	100	100	98	100	100	96
Test#2	100	100	100	100	99	99	99	97
Test#3	100	100	100	100	96	100	100	95
Test#4	100	100	100	100	99	99	99	96
Test#5	100	100	100	100	98	100	100	95
Overall	100	100	100	100	98	100	100	96

Table 7. Recall for micro sampling (%)

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Test#1	100	100	100	100	95	99	99	97
Test#2	100	100	100	100	100	99	99	98
Test#3	100	100	100	100	98	99	99	97
Test#4	100	100	100	100	99	99	99	95
Test#5	100	100	100	100	99	99	99	96
Overall	100	100	100	100	98	99	99	97

Table 8. Recall for random data ratio (%)

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Test#1	100	100	100	100	100	99	99	100
Test#2	100	100	100	100	99	99	99	100
Test#3	100	100	100	100	99	99	99	100
Test#4	100	100	100	100	96	99	99	100
Test#5	100	100	100	100	100	99	99	100
Overall	100	100	100	100	99	99	99	100

Table 9. Recall for CClearner (%)

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
CCLearner	100	98	98	89	28	1	-	-

Table 10. Precision-false alarm rate-accuracy for real data ratio (%)

	Precision		False Alarm Rate		Accuracy
	Clone	Not Clone	Clone	Not Clone	
Test#1	100	92	4	0	99
Test#2	100	89	3	1	99
Test#3	100	90	5	0	99
Test#4	100	87	4	1	99
Test#5	100	89	5	0	99
Overall	100	89	4	0	99

Table 11. Precision-false alarm rate-accuracy for micro sampling (%)

	Precision		False Alarm Rate		Accuracy
	Clone	Not Clone	Clone	Not Clone	
Test#1	100	85	3	1	99
Test#2	100	84	2	1	99
Test#3	100	81	3	1	99
Test#4	100	82	5	1	99
Test#5	100	83	4	1	99
Overall	100	83	3	1	99

Table 12. Precision-false alarm rate-accuracy for random data ratio (%)

	Precision		False Alarm Rate		Accuracy
	Clone	Not Clone	Clone	Not Clone	
Test#1	100	84	0	1	99
Test#2	100	88	0	1	99
Test#3	100	85	0	1	99
Test#4	100	84	0	1	99
Test#5	100	88	0	1	99
Overall	100	86	0	1	99

Table 13. Precision-false alarm rate-accuracy for CCLearner (%)

	Precision		False Alarm Rate		Accuracy
	Clone	Not Clone	Clone	Not Clone	
CCLearner	93	-	-	-	-

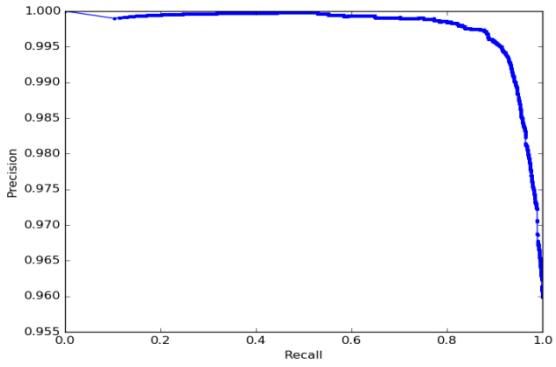

Figure 5. The distribution of precision versus recall on Test#3 with varying thresholds for prediction.

5. INTERPRETATIONS

According to the results, our model has a remarkable success on code clone detection. It has great recall and precision values for both 'clone' and 'not clone' samples. Token sequence representation of method pairs reduces false alarms compared to frequency based representations.

For each experiment, we selected 10000 test samples, but we observed the success of different examples by repeating selecting test samples five times. The execution time of the CNN model, including training and test phases, is about 20 minutes. In addition to this, the time required for tokenization and preparing input features is approximately 35 minutes. The total execution time for the approach we have applied is less than an hour and we think this time period is reasonable for applying a deep learning approach. Considering that our feature vector resembles an image, we observe that CNN is a good algorithm to detect clone.

When we look at the distribution according to clone types, we see that the sample numbers are very different from each other. Although the WT3/4 dominates the dataset, the model gives very successful results in all clone types. We think CNN has the ability to overcome different clone types. Since micro sampling experiment took better and more reliable results, we compared these results with CCLearner [22]. Table 9 and Table 13 show recall, precision, false alarm rate and accuracy results for CCLearner. Some values are missing (represented as -) since they were not reported by Li et al. [22]. While our work and CCLearner have almost equal recall for T1, T2 and VST3 clone types, our results show significantly better results on more sophisticated clone types. For ST3, MT3 and WT3/4, we have 100%, 98% and 99% recall values respectively. However, CCLearner took only 89%, 28% and 1% for these clone types. This is because their training data does not include MT3 and WT3/4 types and

CCLearner algorithm tries to detect these complicated clone types without learning them. We also have better precision on detecting clones. We cannot compare our results with respect to false alarm rate, accuracy and not clone precision since they were not reported by CCLearner.

Copy-paste operations can be done in source code in various ways. For example, the code of two methods can be combined to create a larger single method or a larger method can be divided into two different methods. These cases should also be identified as clone. However, in order to make this determination, it is necessary to create a dataset which takes into account the relations of the methods with each other. If we could create a database that consists of interrelated methods, models like RNN would also be successful [39]. Instead of method granularity, a statement-based model could be built, and the clone detection model could be trained by combining statements in an incremental fashion. Unfortunately, we could not perform such a detailed analysis as our dataset restricts us to construct method pairs for prediction. As a future we plan to analyze code pieces as statements.

6. CONCLUSION

In this work, we propose a clone detection technique which combines tokenization and deep learning practices. The model has good ability to classify java methods as 'clone' and 'not clone'. We took three different sets to train the model and five test sets at each training set. Using these data we trained and tested our CNN. We reported the results comparing the similar approach and saw that our approach has considerable amount of contribution. Compared to prior work, different types of clones (Type 3 and 4) are successfully detected with a recall rate between 98-100% and false positive rate between 2-5%.

In the future, our work can be improved by taking several steps in model construction and dataset. One of them could be varying the dataset size, since we only used a small portion of the whole dataset. By increasing the dataset size, the model could learn the minority clone types better although the majority is still dominated by the MT3 and WT3/4 clone types. A sampling technique would essentially be necessary to keep the balance between different clone types in training set, and therefore, we applied the micro-sampling technique and obtained more successful false alarm rates. The test set should always reflect the real scenario, as in practice when the model is used to predict whether a method pair is clone, this pair is most likely be a MT3 or WT3/4 clone type.

Second, different CNN architecture may be constructed with different layers and initialization parameters. Our CNN model is currently completing training and prediction along with dataset construction less than an hour. We think this is reasonable, but improvements and scalability works can be done. Further, other machine learning methods may be tried instead of CNN.

Another work can be done on feature selection. It is possible to combine CNN and text mining approaches. Acı and Çırak [40] applied both CNN and Word2Vec [24] in order to categorize news articles. Instead of token ID sequences, it may be possible to extract tokenized vectors based on word embeddings so that the place of each token and its relation to the prior and next tokens can be considered. Nevertheless, it is challenging to construct equal sized token lists because of a wide variety of method size. Our future research direction is to combine CNN with text mining approaches such as topic modelling and word embeddings to identify related tokens, and analyze their impact on code clone detection.

REFERENCES

[1] C. K. Roy and J. R. Cordy, "A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools", **4th International Workshop on Mutation Analysis (MUTATION) in 2nd International Conference on Software Testing, Verification, and Validation Workshops**. Denver, Colorado: IEEE Computer Society, 157–166, 1-4 April 2009.

[2] A. Sheneamer and J. Kalita, "Article: A survey of software clone detection techniques," *International Journal of Computer Applications*, 137 (10), 1–21, 2016

[3] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita, "KClone: a proposed approach to fast precise code clone detection", **3rd International Workshop on Software Clones (IWSC)**, 2009

[4] C. K. Roy, J. R. Cordy, and R. Koschke. "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", *Sci. Comput. Program.*, 74(7), 470–495, 2009.

[5] B. Lague, E. M. Merlo, J. Mayrand, J. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process", **IEEE International Conference on Software Maintenance (ICSM)**, 314-321, Oct. 1997.

[6] J. Johnson, "Visualizing textual redundancy in legacy source", **Conference of the Centre for advanced Studies on Collaborative research (CASCON)**, 171-183, 1994.

[7] S. Ducasse, M. Rieger, S. Demeyer, "A language independent approach for detecting duplicated code", **15th International Conference on Software Maintenance (ICSM)**, 109-118, 1999.

[8] C.K. Roy, J.R. Cordy, "An empirical study of function clones in open source software systems", **15th Working Conference on Reverse Engineering (WCRE)**, 81-90, 2008.

[9] B. Baker, "A program for identifying duplicated code", **24th Symposium on the Interface**, **Computing Science and Statistics**, 49-57, 1992.

[10] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering*, 28(7), 654-670, 2002.

[11] Z. Li, S. Lu, S. Myagmar, Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code", *IEEE Transactions on Software Engineering*, 32(3), 176-192, 2006.

[12] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, H. Iida, "SHINOBI: A real-time code clone detection tool for software maintenance", **Technical Report: NAIST-IS-TR2007011**, Graduate School of Information Science, Nara Institute of Science and Technology, 2008.

[13] I. Baxter, A. Yahin, L. Moura, M. Anna, "Clone detection using abstract syntax trees", **14th International Conference on Software Maintenance (ICSM)**, 368-377, 1998.

[14] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones", **29th International Conference on Software Engineering (ICSE)**, 96-105, 2007.

[15] S. Ducasse, M. Rieger, S. Demeyer, "A language independent approach for detecting duplicated code", **15th International Conference on Software Maintenance (ICSM)**, 109-118, 2009.

[16] B. Baker, "On finding duplication and near-duplication in large software systems", **2nd Working Conference on Reverse Engineering,** 86-95, 1995.

[17] R. Wettel, R. Marinescu, "Archeology of code duplication: Recovering duplication chains from small duplication fragments", **7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing,** 8, 2005.

[18] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics", **3rd Working Conference on Reverse Engineering**, 44-54, 1997.

[19] M. White, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, "Toward deep learning software repositories", **IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)**, 334–345, 2015.

[20] B. Can, "LSTM Ağları ile Türkçe Kök Bulma", *Bilişim Teknolojileri Dergisi*, 12(3), 183-193, 2019.

[21] H.K. Dam, T. Tran, T. Pham, "A deep language model for software code", arXiv preprint:1608.02715, 2016.

[22] L. Li, H. Feng, W. Zhuang, N. Meng, B. Ryder, "CCLearner: A Deep Learning-Based Clone Detection Approach", **International Conference on Software Maintenance and Evolution (ICSME),** 249–260, 2017.

[23] C.K. Roy, J.R. Cordy, "Near-miss function clones in open source software: an empirical study", *Journal of Software: Evolution and Process*, 22(3), 165–189, 2010.

[24] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, "Distributed Representations of Words and Phrases and their Compositionality", **26th International Conference on Neural Information Processing Systems**, Nevada, A.B.D., 3111-3119, 2013.

[25] J. Svajlenko, J.F. Islam, I. Keivanloo, C.K. Roy, M.M. Mia, "Towards a Big Data Curated Benchmark of Inter-Project Code Clones", **Early Research Achievements track of the 30th International Conference on Software Maintenance and Evolution (ICSME**) Victoria, Canada, 2014.

[26] Internet: F. Li, J. Johnson and S. Yeung, "Convolutional Neural Networks for Visual Recognation class in Stanford University, 2018, http://cs231n.github.io/convolutional-networks/

[27] N. Davey, P. Barson, S. Field, R. Frank, "The development of a software clone detector", *International Journal of Applied Software Technology*, 1(3/4), 219-236, 1995.

[28] R. Komondoor, S. Horwitz, "Using slicing to identify duplication in source code", **8th International Symposium on Static Analysis (SAS)**, 40-56, 2001.

[29] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," **31st IEEE/ACM International Conference on Automated Software Engineering,** 2016

[30] Internet: ANTLR, http://www.antlr.org

[31] A. Krizhevsky, I. Sutskever, G.E. Hinton, "ImageNet classification with deep convolutional neural networks", **International Conference on Neural Information Processing Systems (NIPS)**, 1106–1114, 2012

[32] K. Simonyan, A. Zisserman, "Very deep convolutional networks for large-scale image recognition", **International Conference on Learning Representations,** 2014.

[33] S.E. Sahin, A. Tosun, "A Conceptual Replication on Predicting the Severity of Software Vulnerabilities", **International Conference on Evaluation and Assessment in Software Engineering (EASE)**, Copenhagen, 2019.

[34] J. Rokui, "Autoassociative Signature Authentication Based on Recurrent Neural Network", **Artificial Intelligence and Soft Computing**, Editors: L. Rutkowski, R. Scherer, M. Korytkowski, W. Pedrycz, R. Tadeusiewicz, J.M. Zurada, Springer, 88-96, 2018.

[35] S. Agarwal, H.S. Sikchi, S. Rooj, S. Bhattacharya, A. Routray, "Illumination-Invariant Face Recognition by Fusing Thermal and Visual Images via Gradient Transfer", **Advances in Computer Vision**, Editors: K. Arai and S. Kapoor, 658-670, 2020.

[36] Internet: Y. LeCun, "Lenet, convolutional neural networks," 2015, Available: http: //yann.lecun.com/exdb/lenet/

[37] Y. Bengio, X. Glorot, "Understanding the difficulty of training deep feedforward neural networks", **13th International Conference on Artificial Intelligence and Statistics (AISTATS),** 249– 256, May 2010.

[38] D. Kingma and J. Ba. "Adam: A method for stochastic optimization", **International Conference on Learning Representations**, 2015.

[39] M. Kızrak, B. Bolat "Derin Öğrenme ile Kalabalık Analizi Üzerine Detaylı Bir Araştırma", *Bilişim Teknolojileri Dergisi*, 11(3), 263-286, 2018.

[40] C. Acı, A. Çırak, "Türkçe Haber Metinlerinin Konvolüsyonel Sinir Ağları ve Word2Vec Kullanılarak Sınıflandırılması", *Bilişim Teknolojileri Dergisi*, 12(3), 219-228, 2019.