Analysis of neural networks efficiency for determining positions of corrupted bytes

Vladislav Lupanov and Sergey Slastnikov
National Research University Higher School of Economics, 20 Myasnitskaya Street, Moscow 101000 Russia
E-mail: velupanov@gmail.com, sslastnikov@gmail.com

Abstract.
A lot of files and data, in general, are transferred throughout the networks. But the data may be corrupted by intrusions or package loss so, the executable files may be marked as non-executable and violate the local network policy. Thus, it’s necessary to detect such files. In this paper, we present a novel method for detecting broken bytes of a file, so the corrupted files may be detected. Also, the positions of wrong bytes might be helpful in restoring the original file content.

This work is devoted to study of modern neural network models applied to detect corrupted bytes of a file problem. Since recurrent neural networks (RNNs) seem to be well suited for such tasks, the main tasks of this work are to analyze the efficiency of popular state-of-the-art RNNs solving the problem mentioned above and to compare the results of different models. We use data consisting of the most popular file types collected from the Internet and manually randomly added noise to that data to test our models. An experiment on this data demonstrates the advantages and disadvantages of the considered models.

1. Introduction
Finding positions of broken bytes in the file and determining if a file has broken bytes at all are very important problems nowadays. Bytes of a file may be changed by intruders or virus, so additional checks may be required if unusual bytes are detected during a file check. Also, this method may be useful to detect if significant bytes of a file are changed, which can lead to errors while opening the file. Moreover, the firewall restrictions and policies may be bypassed by intruders by changing the file header. In addition, detecting positions of broken bytes is helpful while restoring the original file content. It can be used to help restoring the file and to validate the restoration process not damaging the uncorrupted bytes.

The goal of this paper is to study the efficiency of applying existing neural network models to solve the determining wrong bytes positions problem. The main tasks of this paper are study, search and analysis of neural network models, that seem to be suitable for solving the stated problem. Neural networks were chosen because this field becomes more and more popular every day and has a rapid growth. Besides, lots of papers on this theme have already been published and are still been published. Furthermore, there are a lot of successful solutions of such a problem class - when the input and output sizes are dynamic and output is generated or chosen from a huge sample.[1]
2. Bytes embeddings

In this section, we describe and study modern methods of obtaining vector representations of bytes. Most of them used to obtain embeddings for words. So, we decided to try these models too due to the fact that relative positions of a word are taken into account in them. We trained all models to obtain vector representations of bytes and N-gram of bytes. N took the following values: 2, 3, 4, 8, 12 and 16. Immediately check the quality of vectors from trained models is a hard task. Thus, the results of using each model will be calculated later, while using each model to determine broken bytes positions.

2.1. Word2Vec

One of the first with a complicated structure and, perhaps, the most popular among embedding models. Was introduced by T. Mikolov et al. in their work [2]. In our work we use the Continuous Bag Of Words (CBOW) model from the stated work. Because the another model - Skip-gram is better in semantic problems. Fig. 1 shows the model architecture. The model is trying to predict the word using it’s context - the words, that stay near, in a fixed-size window. The windows size is a parameter. In this work, we use a window size equal to five, what means that every byte is predicted using four surrounding its bytes. Instead of words, we used not only bytes but the N-gram concatenations of bytes. Bytes were converted to a hexadecimal system and concatenated to use them in a model. For example, bytes 255 and 15 are converted to ”FF” and ”0F” respectively, then, they are joined to ”FF0F”.

2.2. FastText

Very popular model made by researchers from Facebook Inc. Its use cases are both acquiring vector representations of words and classification. Was introduced by P. Bojanowski et al. in their work [3]. Besides, this model provides functionality to get not only word vectors but vectors for sentences and parts of words. Unlike Word2Vec, which was described earlier, FastText has no dictionary and may calculate embeddings for words, that was not used in the training dataset. In the training process, the model is trying to predict the context of a word by the word themself. Moreover, FastText is using not only words but word parts too: N-grams for N from 3 to 6 are taken. Special symbols are added to words to distinguish them from character N-grams and to understand when the word is over. For example, ”<her>” is a word and ”her” is a 3-gram in word ”<where>”.

So-called negative sampling is used in FastText to achieve better results. Negative examples are words that never appear in current word context, they are used in loss function with opposite sign. That means that the likelihood of dissimilar words are minimized, while the distance between them is maximized. The window size is also a parameter like it was in Word2Vec. In our work, we use a window size equal to 5. Which means that every byte (or concatenated N-gram of bytes) is used to predict four surrounding it bytes (or N-grams). Bytes were processed like in Word2Vec described above.

2.3. Autoencoder

Autoencoder is an artificial neural network that uses unsupervised learning to get effective vector representations of input data, most of all, with dimensions reduction. Was firstly introduced by the authors in their work [4]. The simple architecture is used in the most cases - three-layer dense feed-forward network (three-layer perceptron). Typical Autoencoder is shown in Fig. 2. Many modern authors in their works about text processing, clustering, and classification report about the efficiency of Autoencoders applied to reduce the input dimension and to get vector representations of words. In this work, we use the complicated architecture of a model. Similar models are described in different works about text processing and good results in comparison
with statistical methods of vector dimensions reduction (principal component analysis and other) are shown.[5]

A vector of size 256 is passed through the input layer of the model. Input vector consists of zeros except the one 1 that stays in a position corresponding to a decimal byte value. For example, ‘0F’ is 16 in a decimal system, so the 1 would be in 16th position. This method of data encoding is called one-hot encoding. The Autoencoder model is split into two parts: encoder and decoder. The first part is the encoder: several layers of the model reduce the input shape to 32, which is the shape of vector representation. Next is a decoder part. Decoder recovers the encoded input to the source vector. Thus, the input and output of the model are source bytes encoded in one-hot encoding. And the Autoencoder model is learning to map the input vector to the desired dimension and to translate it back to source input.

Figure 1. The CBOW model architecture. The image was taken from [2].

Figure 2. The Autoencoder model architecture: three-layer perceptron.

3. Model architectures
Many different types of neural networks were proposed by many authors. We tested a lot of models in this work, but we have chosen recurrent neural networks (RNNs) class due to their usability in processing sequences. RNNs seem to be well suited for such tasks because the input and the output of the model are sequences and we need to convert input sequence of bytes to output sequence with probabilities of corrupted bytes in sequence.

Recurrent neural networks - class of neural networks where connections between vertices (neurons) form a directed graph (sequence). That’s why it may process time series and sequences. In this work, we study the possibility of predicting the positions of broken bytes with neural networks, so the use of the RNN class seems to be an efficient way, because of their ability in processing sequences.

LSTM - Long Short-Term Memory is a kind of RNN that has an ability to remember input values for both short and long terms. Each LSTM cell consists of three gates: the input gate, which controls the weight of new value in memory. Next, the forget gate, which controls the
extent of remains the values in the cell. And finally, the output gate which controls the extent of the value in the cell used to compute the output function. Fig. 3 shows the architecture of one LSTM cell.

3.1. Sequence LSTM

The most simple among considered neural network models. The main idea is to pass the windows of size N bytes to the input layer. Since files have large sizes in bytes, it would be an inefficient use of resources to pass the full file to the neural network input layer. In this work, we use the window size from 4 to 256 bytes. The input vectors are bytes of the corrupted file, while the output vectors contain the probabilities for every byte to be broken. In this case, when we have positions of corrupted bytes, it could be additional information for the restoration of source file content using the additional models or utilities. Furthermore, the efficiency of the model will be tested both with different vector representation models and without them. The architecture of naive sequence LSTM model is changed depending on window size and embeddings.

3.2. Stateful LSTM

The main difference between stateful LSTM and common LSTM networks is that neurons states are not reset each time the network get a new input. Which means that the memory is reset only when it is needed. In our work we use different approaches:

- Feed N-grams of bytes to input and reset memory after feeding a file completely
- Feed N-grams of bytes to input and reset memory after feeding M bytes

For example, the input window size is 16 and M is equal to 256, which means that memory is reset every 256 bytes. Fig. 5 presents one of our tested model architecture, where the window size of 4 is fed to the network and states of neurons are remembered. States are reset every 256 bytes which are equal to 16 input windows.
3.3. Seq2seq

Sequence To Sequence (seq2seq) is a complex model that consists of many LSTM cells and has a complicated structure. It was proposed by I. Sutskever et al. in their work [6]. This model became very popular because of machine translation problem. Generally, there are two parts: encoder and decoder. Both encoder and decoder are plenty of LSTM cells. The input vector is fed to the encoder after that to the decoder and finally, the output is generated. A special symbol is used in the training process to mark the end of a sequence. It is also fed to a decoder to show the end of input from an encoder and the beginning of the generation process. Fig. 4 shows the architecture of seq2seq model.

In our work, we use a window size of 4 to 64 bytes as input and the output is a vector with probabilities of each byte to be corrupted. In addition, the ability of feeding the full file and get all wrong positions was also studied. What became possible due to seq2seq architecture. The model takes a lot of computing resources if we pass the whole file through it, so we discarded this approach. Fig. 6 presents the simplified example of the used in this work seq2seq model input and output.

![Figure 5](#) Stateful LSTM with window size 4 input and output example. State is reset every 256 bytes.

![Figure 6](#) Seq2seq model input and output example. Window size is equal to 4, end of stream 'symbol' is 'EOS'.

4. Results

To compare the quality of different recurrent neural network models and approaches for getting a vector representation of bytes we use the same data on each combination. Also, we use embeddings trained on another file types to understand the impact of knowing the file extension on the result. Moreover, we try to train vector representation models on all file types to avoid the necessity of knowing the type of file. We try different models described above to understand if the N plays role while using N-grams for getting embeddings. Also, we change the window size to understand the impact of it on the quality of a model.

4.1. Used data

We use data collected from the Internet using Google search by map and extensions. Also, we processed a large dataset of imaged called "ImageNet" which was introduced as a challenge in [7]. All files were processed and a noise was added: we randomly change every byte of a file with probability $p$, where $p$ are: 0.01, 0.05, 0.10, 0.15 and 0.30. To understand the training size it is better to provide not a count of files but the total size of datasets of each file type. The numbers of bytes are shown in Table 1. All the data is shuffled split into a training and test sets. The test size is 20% of dataset size.

4.2. Comparison of results for different file types

Table 2 illustrates results of best models applied for the stated problem for different file types. For "exe" file extension models scores are slightly better than a random pick. So, some upgrades
Table 1. Distribution of file types

| File type | Bytes count, million |
|-----------|---------------------|
| EXE       | 3633                |
| JPG (JPEG)| 3200                |
| MP3       | 2715                |
| PDF       | 4075                |

Table 2. Precision of models for different file types

| Error rate | File type |
|------------|-----------|
|            | jpg      | pdf      | exe      | mp3      |
| 1%         | 0.9951    | 0.9928   | 0.9900   | 0.9932   |
| 5%         | 0.9732    | 0.9711   | 0.9501   | 0.9801   |
| 10%        | 0.9291    | 0.9246   | 0.9004   | 0.9202   |
| 15%        | 0.8948    | 0.8788   | 0.8505   | 0.8890   |
| 20%        | 0.8744    | 0.8592   | 0.8009   | 0.8530   |
| 30%        | 0.7782    | 0.7523   | 0.7013   | 0.7637   |

are needed to achieve better results. We got an averaged 0.8923 precision and 0.9991 recall. Such precision means that if you take any byte that is marked by a model as corrupted, this is a probability for a byte to be corrupted really. Similarly for recall: it means the probability that a marked as a correct byte is actually a corrupted one.

5. Conclusion

In this paper we studied the quality of different recurrent neural network applied to determine broken bytes positions. Also, we described what changes are needed to apply these models to determine the positions of corrupted bytes. In addition, we considered the most popular models for getting vector representation of the word and used them to get an embeddings for bytes and N-grams of bytes. All needed transformations to feed the bytes to the vector representation models are provided. Next, comparison of getting embeddings using different approaches is shown in plots. As well as tables showing the quality of all described neural networks applied to the stated problem is demonstrated. Finally, all sort of combinations of vector representation models and RNNs were compared and results are shown in the table.

To sum it up, recurrent neural networks proposed in this paper are well suited for the file analysis and processing due to their mechanisms which allow variable size inputs. The solution of the determining the corrupted bytes positions problem may be used in such applications, as network security, file validation, file recovery and so on.

References

[1] Raff E, Barker J, Sylvester J, Brandon R, Catanzaro B and Nicholas C 2017 ArXiv e-prints (Preprint 1710.09435)
[2] Mikolov T, Chen K, Corrado G S and Dean J 2013 CoRR abs/1301.3781
[3] Bojanowski P, Grave E, Joulin A and Mikolov T 2017 Transactions of the Association for Computational Linguistics 5 135–146 ISSN 2307-387X
[4] Hinton G E and Salakhutdinov R R 2006 Science 313 504–507 ISSN 0036-8075 (Preprint http://science.sciencemag.org/content/313/5786/504.full.pdf)
[5] Xie J, Girshick R and Farhadi A 2016 Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 ICML’16 (JMLR.org) pp 478–487
[6] Sutskever I, Vinyals O and Le Q V 2014 Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 NIPS’14 (Cambridge, MA, USA: MIT Press) pp 3104–3112
[7] Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, Huang Z, Karpathy A, Khosla A, Bernstein M, Berg A C and Fei-Fei L 2014 ArXiv e-prints (Preprint 1409.0575)