Utilizing artificial neural networks for Ethernet II error correction.

Marek Szymański, Mateusz Nurczyński

Abstract

Based on the assumption, that live Internet communication is, in it’s nature, self-similar, we asserted that it is possible to build a machine learning model, that would identify and correct errors in Internet transmissions.

Currently there are no error correction methods in use, that would attempt to employ a statistical model analyzing the very data being transmitted.

To verify the basic assumptions of this hypothesis (that self-similarity of the Internet traffic can be recognized by a machine learning model and used to correct errors), we attempted to train an artificial neural network to correct single-bit errors in Ethernet II frames captured from live Internet traffic.

We identified three potential approaches, that might be utilized: autoencoders (trained exclusively on undamaged frames), classifiers (with as many classes as there is bits in the frame – such a model would simply flag bits that it thinks to be damaged) and direct mapping (model receiving a damaged frame and returning an undamaged one).

We found, that our assertion (that machine learning models could be used to correct transmission errors based on the data being transmitted) to be true in a strictly factual sense. However, low effectiveness of this method of error correction makes it unfeasible as a standalone solution.

We also found, that of the three aforementioned approaches only the Direct Mapping is effective and only on small data patches (up to 100 bytes), therefore large frames have to be fragmented.

We found limited evidence of models adapting to specific Ethernet II frame categories (protocols, payloads), thus we suspect that more specialized models could achieve higher effectiveness.

We suggest, that a simple classifier could be used to differentiate fragments between different categories and feed data to specialized models. This approach could potentially be used in conjunction with other correction methods.

Introduction

Self-similar Internet

Self-similar nature of the Internet traffic has been demonstrated by multiple researchers: ,,We also find a notable self-similarity feature of the autocorrelations in the data and its aggregates, in all the cases studied.”[1].

I’s also been found, that Ethernet traffic demonstrates some degree of fractal nature: ,,the critical characteristic of this self-similar traffic is that there is no natural length of a "burst": at every time scale ranging from a few milliseconds to minutes and hours, similar-looking traffic bursts are evident”[2]

Such findings are not surprising, since across all abstraction layers there is a limited number of Internet protocols in wide use, which naturally leads to certain elements (like headers of high-level protocols or even payloads of widely used network applications) repeating frequently in transmissions. Furthermore, different protocols are similar in nature.

Possible implications for error correction

However, the possibility of building a statistical model based on those characteristics have never been explored as a possible way to identify transmission errors.

Even within a single Ethernet frame there is a multitude of dependencies across all network abstraction layers. For example, information provided in an HTTP header (like language and file type) will correspond to the payload.

We asserted, that those dependencies could be used to build a deep learning model, that would allow to correct network transmission errors.

We performed a series of tests on live Ethernet II traffic, attempting to train artificial neural networks of various sizes and architectures for correcting single-bit transmission errors. The purpose was to verify whether this method of error correction is at all feasible.

This document is a synthesis of all our findings.

1 D. Chakraborty, A. Ashir, T. Suganuma, G. Mansfield Keeni, T. K. Roy, N. Shiratori ,,Self-similar and fractal nature of Internet traffic”

2 W. E. Leland, M. S. Taqqu, W. Willinger, D. V. Wilson ,,On the self-similar nature of Ethernet traffic”

Methods

Below we outline in great detail how we performed the aforementioned tests.

Data gathering and preparation

Below is a description on how we gathered and transformed the data for our model.

Capturing

We collected the data for our analysis using version 4.0.8 of open-source Wireshark software, running on a 64-bit x86 machine under a Unix-based operating system (Fedora Linux Workstation 37). All communication was captured in a small 1Gb Ethernet LAN network, while the computer was being used normally.

We captured over 220 000 Ethernet II frames of various sizes and payloads. We could only capture valid (non-damaged) frames and without their CRC check-sum.

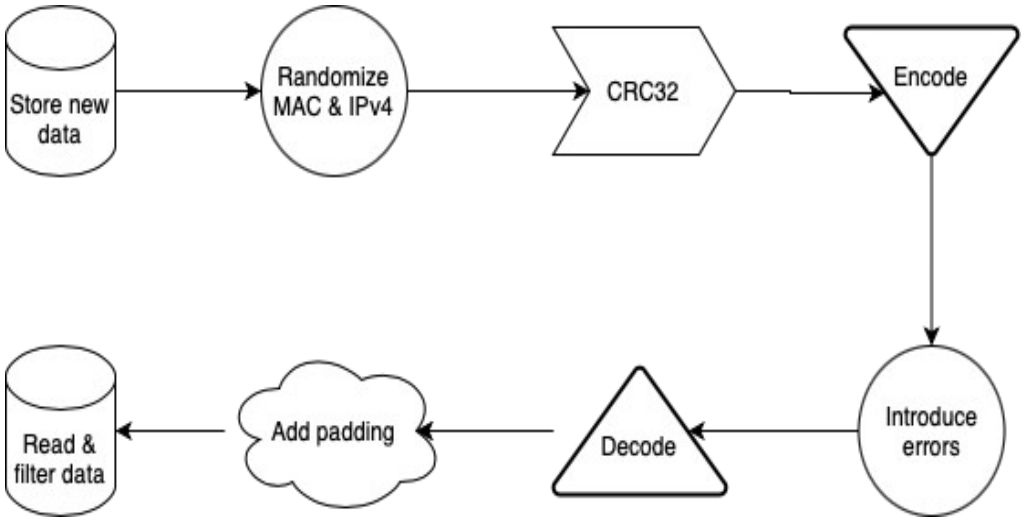
Filtering

Since our purpose was to verify whether this way of error correction is at all feasible, we decided to simplify the problem by only including frames carrying IPv4 packets (210 000 out of 220 000 frames).

General outline of data preprocessing

We begin with all the captured non-IPv4 frames already filtered out

1. MAC and IP are randomized
2. CRC is calculated and appended at the end of the each frame
3. Each frame (with CRC) is encoded using 8b/10
4. Single-bit errors introduced into the encoded frame
5. Each frame is decoded back from 8b/10b, the single-bit errors propagate into entire octets changed.
6. The frame is padded with 0s to the desired length (in bytes) and the CRC moved to the end



1. Address randomization

Being collected form only one machine, operating in only one, fixed network configuration all of our frames had the same two MAC addresses in them, as well as a subset of similar IP addresses. In order to avoid contaminating the model with that information, we decided to randomize all MAC and IP addresses in our data. Therefore both types of addresses in all processed frames were replaced with randomly generated bytes.

2. Cyclic Redundancy Check (CRC)

As the frames captured by the Wireshark software lacked the FCS field we had to calculate it ourselves. Ethernet II uses a 32-bit variant of the Cyclic Redundancy Check (CRC). Our implementation uses the popular reverse polynomial 0xEDB88320 and pre-generated lookup table to compute the check-sum according to the standard CRC algorithm.

3. Encoding

In order to better imitate the way transmission errors are introduced into frames in real-life conditions we have decided on the 8b/10b encoding, one still in use in modern Ethernet networks, even if less popular than the 64b/66b. The 8b/10b line code maps each 8-bit word to a 10-bit symbol in such a way to avoid chains of more than 5 ones or zeros and to ensure the disparity between the number of ones and zeros in strings of bits longer than 20 does not exceed 2. As the encoding is usually handled on the hardware level we have created our own software implementation.

4. Introducing errors into the frames

In every encoded frame at least 1 bit was changed, with a 0.1 chance of second bit also being changed. The positions of changed bits were randomly generated with

uniform distribution covering the entire frame (including the appended CRC checksum).

5. Error propagation

The single-bit errors introduced into 10-bit symbols of the 8b/10b encoded frame lead to those symbols being decoded into different 8-bit words than they were originally encoded from. This may result in more bits being changed in the decoded frame.

6. Buffering and CRC position change

All frames that were below maximum size had to receive a buffer of zeroes at the end in order to fit into the fixed-size input layer of a neural network. Since the check-sum may prove useful for our model to correct the error, we needed it placed in a fixed position, not matter the frame size. Therefore, after decoding, we put it at the end of the buffered frame:

Frame header and payload	Buffer (zeroes)	CRC

Addendum: pseudo-random number generator used

We have used the mt19937 Mersenne Twister pseudo-random generator from the C++ standard library.

Prepared datasets

In order to conduct our tests, after preprocessing the frames as described above, we prepared several datasets of different formats. All the datasets are listed, marked and described below. All datasets are comprised of a training dataset and a testing dataset.

D-EXOR-L – dataset mapping damaged frames to XORs

Each data point consists of a damaged Ethernet frame given as a binary input vector consisting of 12 144 elements (8 bits multiplied by maximum standard Ethernet II frame size – 1518). As a desired network output we give an output vector, that is a bit-wise XOR of the damaged frame with the correct frame.

Assuming that the captured frame without errors (the input vector) is:

1 1 0 0 1 0 1 0

And the frame after introducing errors is:

1 0 0 0 1 1 1 0

Then the output vector is:

0 1 0 0 0 1 0 0

Training set: 180 000 frames

Test set: 30 000 frames

D-EXOR-S – dataset mapping damaged frames, that are 100 byte long or shorter, to XORs

This dataset is analogous to D-EXOR-L, except it only contains frames that are 100 byte long or shorter. It means, that both output and input vectors are 800 element long.

Training set: 71 000 frames

Test set: 14 000 frames

D-CC-S – autoencoder dataset

Each data point is comprised of one correct (ie. without errors) frame, given both as the input vector and the output vector.

Just as D-EXOR-S, this dataset only consists of frames that have 100 bytes or less.

Training set: 71 000 frames

Test set: 14 000 frames

D-EC-L – dataset mapping damaged frames to correct frames

This dataset is analogous to D-EXOR-L, except instead of the XOR vector, an output vector containing the correct frame is given.

Training set: 180 000 frames

Test set: 30 000 frames

D-EC-S – dataset mapping damaged frames, that are 100 byte long or shorter, to correct frames

This dataset is analogous to D-EC-L, except it only contains frames that are 100 byte long or shorter. It means, that both output and input vectors are 800 element long.

Training set: 71 000 frames

Test set: 14 000 frames

D-EC-F32 – dataset containing 32 byte long fragments of frames

Input and output vectors are built in the same way was in D-EC-L and D-EC-S, but the CRC checksum is not included and instead of entire frames, each data point is a 32-byte long fragment.

Training set: 100 000 fragments

Test set: 100 000 fragments

D-EC-F100 – dataset containing 100 byte long fragments of frames

Input and output vectors are built in the same way was in D-EC-L and D-EC-S, but the CRC checksum is not included and instead of entire frames, each data point is a 100-byte long fragment.

Training set: 100 000 fragments

Test set: 100 000 fragments

Tests

This section of the document is a detailed explanation of all experiments conducted on the aforementioned datasets, using artificial neural networks.

We identified three potential approaches to the problem:

- autoencoders (trained exclusively on undamaged frames)
- classifiers with as many classes as there is bits in the frame – such a model would simply flag bits that it thinks to be damaged
- direct mapping (model receiving a damaged frame and returning an undamaged one).

Those approaches will be discussed further in the Results chapter.

We employed the PyTorch framework for Python in order to conduct the tests.

In all tests we used MSE as the loss function, unless specified otherwise.

In all tests learning rate was set to 1e-3, unless specified otherwise.

The optimizing algorithms used were Adam and Adamax.

Test 1 – Classifier on large frames

Dataset: D-EXOR-L

We attempted to train models ranging from 1 to 4 deep layers, all uniform (12144 nodes) with Sigmoid activation function on the output layer and Sigmoid or ReLU functions on other layers.

We attempted to run training on batch sizes 400, 500, 1000 and 5000 for as many as 100 epochs.

Regardless of hyperparameters after just the first few epochs models learned to set all output bits to 0.

We also attempted to run the training with cross-entropy function instead of MSE, with similar results.

Test 2 – Classifier on smaller frames (no larger than 100 bytes)

Dataset: D-EXOR-S

We attempted to train a model with one deep layer (800 nodes). We used Sigmoid as an activation function for the output layer. We tried both ReLu and Sigmoid on other layers.

We attempted to run training on batch sizes 400, 500, 1000 and 5000 for as many as 100 epochs.

Regardless of hyperparameters after just the first few epochs models learned to set all output bits to 0.

We also attempted to run the training with cross-entropy function instead of MSE, with similar results.

Test 3 – Autoencoder with 3 deep layers

Dataset: D-CC-S

We attempted to train models with 3 deep layers (sizes: 100, 10 and 100 nodes) and 7 deep layers (sizes: 400, 200, 100, 50, 100, 200 and 400 nodes)

We used Sigmoid function for the output layer and we tried both Sigmoid and ReLU for other layers.

We used the following batch sizes: 100, 500, 1000, 5000, 10000 for as many as 100 epochs.

Test 4 – Direct mapping of damaged frame onto the correct frame for large frames

Dataset: D-EC-L

We ran this test on similar network architectures as in Test 1. After 1 epoch it achieved loss of approximately 0.26 and did not learn any further.

Test 5 – Direct mapping for small frames (no larger than 100 bytes)

Dataset: D-EC-S

We attempted to train models with between 1 and 4 deep layers (different configurations of the following layer sizes: 800, 1600 and 2400 nodes).

We used Sigmoid for the output layer and different combinations of ReLU and Sigmoid for other layers.

Ultimately the best performing model had 1 deep layer with 1600 nodes and ReLU activation function on batch size 400 (we also tried 500, 1000 and 5000) after 50 epochs of training.

Test 6 – Direct mapping for small fragments of frames (32 bytes)

Dataset: D-EC-F32

We attempted to train a model with 1 deep layer of 1024 nodes with ReLU as an activation function.

We used Sigmoid activation function for the output layer.

Best results were achieved for the batch size of 1000 with loss of approximately 5e-4.

There was also an interesting case, when the network achieved good overall results, despite relatively high loss (0.02). This will be discussed further in the Results chapter.

Test 7 – Direct mapping for small fragments of frames (100 bytes)

Dataset: D-EC-F100

We attempted to train models with between 1 or 4 deep layers of 800 nodes each, with ReLU as an activation function.

We used Sigmoid activation function for the output layer.

Best results were achieved for the batch size of 1000 with loss of approximately 1e-3 for the smaller model and 0.19 for the larger one.

Results

This section is a brief synthesis of achieved results, broken down by approach.

Autoencoders

Test 3.

General idea

We attempted to train autoencoders of several different architectures for recreating undamaged 100-byte long frames. The idea was, that an autoencoder after seeing multiple undamaged frames could then recreate a correct frame based on a damaged one.

Results

None of the autoencoders we trained was able to successfully restore even a single frame. We did not pursue this idea further.

Direct frame restoration

Tests 4, 5, 6 and 7

General idea

This is the most direct approach – we attempted to train a neural network to take a damaged frame as input and return a fixed frame.

Results

We had no success with this approach on D-EC-L dataset.

We had limited success on D-EC-S dataset, so with frames no longer than 100 bytes.

The best results achieved on the training dataset were as follows:

- On average output frames had around 4.5 incorrect bits (out of 800). This is significantly worse than frames given as input – average of 2.3 incorrect bits per frame.
- The model fixed 29 out of 13857 test frames (approximately 0.2%)

From that we may confer, that the model fixed errors in some frames, while further damaging other frames.

Average error number per frame is higher after using our model, but those errors are not evenly distributed, as was the case before.

Also worth noting is that in the original testing dataset there was around 4000 frames, in which only 1 bit was damaged. Meanwhile in the output of the model described above there were only 335 such frames. This implies, that our model only learned to fix a specific type of frame and was further damaging all other types.

Since fixing entire large frames was not possible we attempted to divide them into fragments and process through a smaller network.

The best results achieved for 32-byte fragments (D-EC-F32 dataset) were as follows:

- On average output fragments had around 5.33 damaged bits (compared to 0.11 in the original fragments)
- The model fixed (or recognized as undamaged) 1383 of all (both damaged and undamaged) 100000 fragments (approximately 1.3%)
- Out of 4786 damaged fragments the model fixed 920 (approximately 19.2%)

It is worth noting, that this result was a significant outlier and we were not able to replicate it.

Results for the second-best model are as follows:

- On average output fragments had around 0.11 damaged bits (the same as original fragments)
- The model fixed (or recognized as undamaged) 95008 out of all 100000 fragments (approximately 95%)
- Out of 4786 damaged fragments the model fixed 30 (approximately 0.6%)

The second model also has much lower loss (MSE), despite fixing less fragments.

I appears, that the better the model gets at fixing damaged fragments, the worse it is at recognizing which ones are not damaged at all.

The best results achieved for 100-byte fragments (D-EC-F100 dataset) are as follows:

- On average output fragments had around 0.47 damaged bits (compared to 0.29 in the original fragments)
- Out of all 100000 fragments the model fixed (or recognized as undamaged) 73252 fragments (approximately 73.3%)
- Out of 12686 damaged fragments the model fixed 459 (approximately 3.6%)

Classifiers

Tests 1 and 2

General idea

We essentially attempted to create a classifier, with as many classes as bits in a frame. The network for supposed to flag the bits where errors are located. For that we used D-EXOR-L and D-EXOR-S datasets.

This approach could potentially be better that direct restoration, since the neural network’s capacity is not wasted on replicating undamaged bits.

Results

We had no success whatsoever with this approach. Models were not able to fix even a single frame (instead flagging all bits as undamaged).

Discussion

Conclusions and findings

With respect to our assertions detailed in the Introduction to this document, we conclude that:

- To an extremely limited extent it is possible to build a statistical model for correcting transmission errors based on analyzing dependencies within the contents of singular Ethernet II frames.
- Specifically, deep learning algorithms are able to correct Ethernet II transmission errors based on such analysis.
- Autoencoders we found to be absolutely ineffective in restoring units of Internet traffic.
- Classifiers we found to be absolutely ineffective in restoring units of Internet traffic.

- The most effective neural network architecture in Ethernet error correction, we found to be a model with 1 deep layer, with Sigmoid function for the output layer and ReLU activation function for other layers, that that directly maps a damaged frame to an undamaged frame.
- We found, that repairing large (1500 bytes) data units is not feasible at all. Therefore large frames (or other data units) need to be divided into shorter fragments.

Overall, we found our assertion to be technically true. However currently the extremely limited effectiveness of this method of error correction makes it unfeasible for any practical application, as a standalone correction algorithm.

Recommendations and possibilities for further research

In general, error correction assisted by content analysis is possible, but not viable in this form as a standalone solution.

There is evidence, that our models adapted to a specific category (or categories) of frames.

Therefore could likely be improved, if specialized models were used to handle different categories of data units (which would probably need to be shorter fragments of frames, datagrams or packets).

A possible solution could include a simple classifier with a few classes to divide data units into categories, that would be handled by specialized correcting models.

It could be possible to utilize this method in conjunction with other error correction algorithms and in this way improve effectiveness of those algorithms.

Another idea that could be explored is utilization of recursive neural networks, which could potentially recognize dependencies between multiple subsequent data units of a specific category.