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 model to correct single-bit errors in Ethernet II frames captured from live Internet traffic.

<here we will describe our results once we have them>

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]

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"

Such finding 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.

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 with 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, 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.

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.

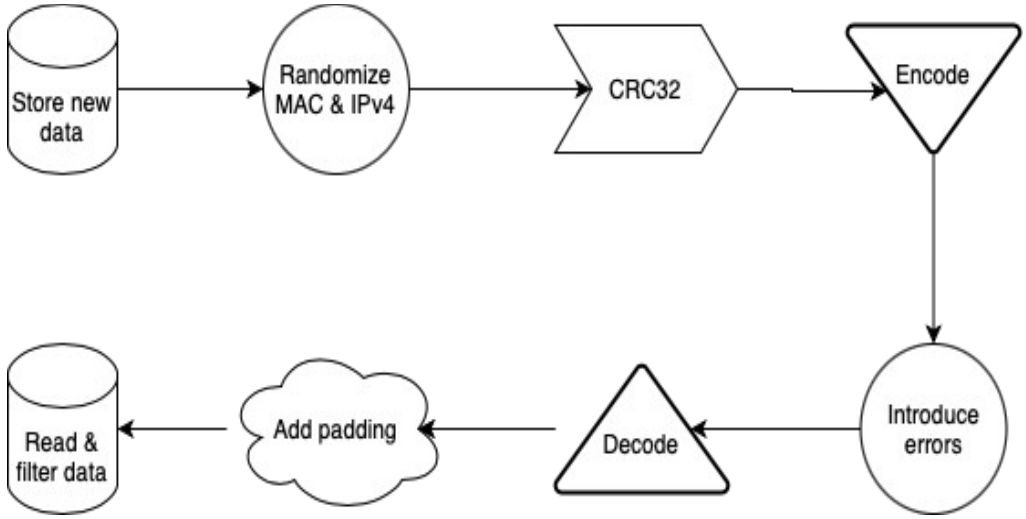
Filtering

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 10-bit bytes 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. As such both types of addresses in 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 checksum 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 0.1 chance of second bit also being changed. The positions of changed bits were randomly generated with uniform distribution covering entire frame (with the appended CRC).

5. Error propagation

The single-bit errors introduced into the select 10-bit symbols of the 8b/10b encoded frame lead to the 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 descirbed 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

Test set: 14 000

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

Test set: 14 000

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

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

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

Training set: 71 000

Test set: 14 000

D-EXOR-F4 – dataset containing 4 byte long fragments of frames

Input and output vectors are build in the same way as in D-EXOR-L and D-EXOR-S, but the CRC check-sum is not included and instead of entire frames, each data point is a 4 byte long fragment.

That makes for 32 element long input and output vectors.

Training set: 100 000

Test set: 100 000

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

Analogous to D-EXOR-F4, except frame fragments are 32 byte long, instead of 4 byte long.

Training set: 100 000

Test set: 100 000

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

Analogous to D-EXOR-F4, except frame fragments are 100 byte long, instead of 4 byte long.

Training set: 100 000

Test set: 100 000

Tests

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

Test 1 – Classifier on large frames

Dataset: D-EXOR-L

Model:

- Linear(12144, 12144) + Sigmoid / ReLU [IN]
- Linear(12144, 12144) + Sigmoid / ReLU
- Linear(12144, 12144) + Sigmoid / ReLU
- Linear(12144, 12144) + Sigmoid / ReLU
- Linear(12144, 12144) + Sigmoid [OUT]

Loss function: MSE

Optimizer: Adamax (Adam)

Learning rate: 1e-3

Epochs: 20 (50, 100)

Batch size: 1000 (100, 500, 5000, 10000)

Result: after only about 2000 frames the model achieves the loss of about 1.8e-4 and subsequently stops any learning whatsoever

Test 2 – Autoencoder with 3 deep layers

Dataset: D-CC-S

Model:

- Linear(800, 100) + ReLU [IN]
- Linear(100, 10) + ReLU
- Linear(10, 100) + ReLU
- Linear(100, 800) + Sigmoid [OUT]

Loss function: MSE

Optimizer: Adam

Learning rate: 1e-3

Epochs: (20, 50, 100)

Batch size: 1000 (100, 500, 5000, 10000)

Result: the autoencoder was unable to properly recreate even single frame, the learning of the network would not progress

Test 3 – Autoencoder with 7 deep layers

Dataset: D-CC-S

Model:

- Linear(800, 400) + Sigmoid / ReLU [IN]
- Linear(400, 200) + Sigmoid / ReLU
- Linear(200, 100) + Sigmoid / ReLU
- Linear(100, 50) + Sigmoid / ReLU
- Linear(50, 100) + Sigmoid / ReLU
- Linear(100, 200) + Sigmoid / ReLU
- Linear(200, 400) + Sigmoid / ReLU
- Linear(400, 800) + Sigmoid [OUT]

Loss function: MSE

Optimizer: Adam

Learning rate: 1e-3

Epochs: (20, 50, 100)

Batch size: 1000 (100, 500, 5000, 10000)

Result: the autoencoder was unable to properly recreate even single frame, the learning of the network would not progress

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

Dataset: D-EC-S

Model:

- Linear(800, 800) + ReLU [IN]
- Linear(800, 800) + Sigmoid [OUT]

Loss function: MSE

Optimizer: Adam (Adamax)

Learning rate: 1e-3 (1e-1, 1e-2, 1e-4, 1e-5)

Epochs: 50 (20, 100)

Batch size: 400 (100, 500, 1000, 5000, 10000)

Result:

1. this model managed to reliably recreate about 200 full frames without a single wrong bit (about 170-180 after 20 epochs, up to 215 after 100 epochs)

2. it also achieved about 3800-3900 frames recreated with single wrong bit

3. on average the frames recreated by the model had about 3 bits wrong compared to clean frames

4. the model reached lower boundary of loss around about 3e-3, not progressing further even after more epochs

Additional info: surprisingly adding further hidden layers to the model actually lead to significant decrease in performance – with only one Linear(800, 800) or bigger the model would properly recreate only about 90 frames while adding 2 or more hidden Linear modules led to no frames at all being recreated properly)

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

Dataset: D-EC-S

Model:

- Linear(800, 2400) + ReLU [IN]
- Linear(2400, 800) + Sigmoid [OUT]

Loss function: MSE

Optimizer: Adam

Learning rate: 1e-3

Epochs: 50 (20)

Batch size: 400 (500, 1000, 5000)

Result:

1. this model managed to reliably recreate about 240 full frames without a single wrong bit

2. it also achieved about 4000 frames recreated with single wrong bit

3. the model reached lower boundary of loss around about 3e-3, not progressing further even after more epochs

Additional info: while this model achieved marginally better in the number of correctly recreated frames it also proved significantly more volatile in training – in more than half cases after training the model would prove less potent than the previous one

Test 7 – Classifier on smaller frames (no larger than 100 bytes)

Dataset: D-EXOR-S

Model:

- Linear(800, 800) + ReLU [IN]
- Linear(800, 800) + Sigmoid [OUT]

Loss function: MSE

Optimizer: Adam

Learning rate: 1e-3

Epochs: 20 (50, 100)

Batch size: 1000 (400, 500, 1000, 5000)

Result:

1. this model managed to reliably recreate about 250 full frames without a single wrong bit (about 170-180 after 20 epochs, up to 215 after 100 epochs)

2. it also achieved about 4000 frames recreated with single wrong bit

3. similarly to previous model trained on the D-EXOR datasets this one reached it's minimal loss of about 3e-3 after only 5000 frames (in 1st epoch) subsequently not progressing further

Additional info: again adding further hidden layers to the model decreased it's performance

Test 8 – Classifiers for small fragments of frames (4, 32 and 100 bytes)

Dataset: D-EXOR-F4

Model:

- Linear(32, 128) + ReLU [IN]
- Linear(128, 128)
- Linear(128, 32) + Sigmoid [OUT]

Loss function: MSE

Optimizer: Adam

Learning rate: 1e-3

Epochs: 20

Batch size: 1000

Dataset: D-EXOR-F32

Model:

- Linear(256, 2048) + ReLU [IN]
- Linear(2048, 2048)
- Linear(2048, 256) + Sigmoid [OUT]

Loss function: MSE

Optimizer: Adam

Learning rate: 1e-3

Epochs: 20

Batch size: 1000

Dataset: D-EXOR-F100

Model:

- Linear(32, 128) + ReLU [IN]
- Linear(128, 128)
- Linear(128, 32) + Sigmoid [OUT]

Loss function: MSE

Optimizer: Adam

Learning rate: 1e-3

Epochs: 20

Batch size: 1000

Result: none of the fragments with errors in them were properly recreated by any of the models used

Results

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

Autoencoders

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

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 limited success with this approach on D_EC_S dataset, so with frames no longer than 100 bytes.

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

- On average output frames had around 3 incorrect bits (out of 800). This is marginally worse than frames given as input – average of 2.5 incorrect bits per frame.
- The model fixed entirely 215 out of 14102 test frames (approximately 1.5%)
- The model almost fixed approximately 3900 frames, with only one bit off (compared to an entirely undamaged frame).

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.

Classifiers

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 datasets.

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

Results

<big frames here>

On D_EXOR_S dataset (so frames 100-byte long and shorter) we had limited success.

Testing results of our best model are as follows:

- On average output frames had 2.2 incorrect bits (so approximately 12% less than input frames).
- The model fixed completely 245 out of 14102 testing frames (approximately .

We also attempted this approach 4-byte, 32-byte and 100-byte on fragments of larger frames (respectively D_EXOR_F4, D_EXOR_F32 and D_EXOR_F100 datasets).

Irrespective of fragment size we had no success in training those models – they were unable to fix even a single frame fragment.

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.
- The most effective neural network architecture in Ethernet error correction, we found to be a classifier with as many classes as there is bits in the frame.

Overall, we found our assertion to be technically true. However currently the extremely limited effectiveness of this method of error correction makes it ineffective 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.

Results could likely be improved, if specialized models were used to handle different categories of frames, packets, protocols and payloads.

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 recognized dependencies between multiple subsequent frames of a specific category.