Laboratory work 3: Cyber attack of type 'perl' recognition with two-layer-perzeptron on NLS KDD data-set

Daniel Paliura

16 12 2020

Purpose

To develop software for realisation of neural networks of types two-layer-perzeptron (TLP) and probability neural network (PNN) dedicated to recognition of cyber attacks, whichs signatures given in data base NLS KDD or KDD-99.

Task

Recognition of cyber attack of type 'perl' using TLP. Show NN error after 1, 50 and 100 epochs.

Introduction

As data base was selected NLS KDD. In this report it will be called 'data-set' or simply 'set'. It was downloaded from Git-hub repository (clickable). To provide data into neural network (NN) data-set must contain only numbers, not symbolic data, so symbolic fields of data-set must be encoded. Also all fields should be normalized to provide stable data into NN. Also there is **restriction** for encoding: processed data-set must contain number of fields equal to number of fields in initial data-set, so one-hot encoding not allowed to use even it is a single way to encode data correctly. And also NN must be written by hand, without using special libraries providing NN solutions.

Solution

Neural network implementation

To write specified in **task** NN I used Python language (v.3.8.3) with next libraries:

- numpy 1.19.2
- pandas 1.1.4

TLP implemented as `class TLP` and has next attributes and methods:

- `sigmoid(X)` - static method - logistic function $\sigma = \frac{1}{1+e^{-x}}$
- `sigmoid_deriv(X)` - static method - derivative of logistic function (not used)
- `feed_forward(self, X, show_progress=False)` - method wich returns TLP output for input `X`. `X` expected to be numpy array and can be one-dimensional array in case single input and two-dimensional numpy array in case multiple inputs. In this case different inputs must be given as rows of array. Parameter `show_progress` indicates whether progress of execution must be shown (prints percentage when it incremented). Actual only for multiple inputs.

- `_feed_forward(self, X)` - protected method which implements neurons activation. This function takes single input and called is from `feed_forward`'s body.
- `back_propagation(self, X, Y, train_rate=2)` - method for single weights correction by back propagation principle. It takes single input vector `X` and according output *vector* `Y` of expected output. `train_rate` is parameter of training velocity, it is just coefficient before derivative in fomulae of deltas.
- `back_prop_epoch(self, X, Y, train_rate=2, show_progress=False)` - method wich takes matrix `X` of multiple training inputs and matrix `Y` of according outputs. This method just calls `back_propagation` method for each train input-output pair. Parameter `train_rate` provides into specified method. `show_progress` is analogue for eponymous parameter in `feed_forward` method.

I built TLP with structure: 41 neuron in

Data processing

Data-set NLS KDD has 42 fields, where 42-th one is class of cyber attack ('normal' if no cyber atack). It contains tree fields (2-th, 3-th, 4-th) of type 'symbolic'. Analysis of set wasn't performed due to **restriction**. Also it wasn't provided for this laboratory work. Data-set was already devided into train set, validation subset (20% of train set) and test set. Sets are given without field names, but field names given in other file. I added names to data frame, which I processed and save processed data into files including field names.

Summary of NLS KDD train subset:

```
##     duration       protocol_type     service           flag
##  Min.   :    0.0   icmp:  8291   http    :40338   SF     :74944
##  1st Qu.:    0.0   tcp :102688   private :21853   S0     :34851
##  Median :    0.0   udp : 14993   domain_u: 9043   REJ    :11233
##  Mean   :  287.1                 smtp    : 7313   RSTR   : 2421
##  3rd Qu.:    0.0                 ftp_data: 6859   RSTO   : 1562
##  Max.   :42908.0                 eco_i   : 4586   S1     :  365
##                                  (Other) :35980   (Other):  596
##    src_bytes           dst_bytes              land           wrong_fragment
##  Min.   :0.000e+00   Min.   :0.000e+00   Min.   :0.0000000   Min.   :0.00000
##  1st Qu.:0.000e+00   1st Qu.:0.000e+00   1st Qu.:0.0000000   1st Qu.:0.00000
##  Median :4.400e+01   Median :0.000e+00   Median :0.0000000   Median :0.00000
##  Mean   :4.557e+04   Mean   :1.978e+04   Mean   :0.0001985   Mean   :0.02269
##  3rd Qu.:2.760e+02   3rd Qu.:5.160e+02   3rd Qu.:0.0000000   3rd Qu.:0.00000
##  Max.   :1.380e+09   Max.   :1.310e+09   Max.   :1.0000000   Max.   :3.00000
##
##      urgent               hot          num_failed_logins    logged_in
##  Min.   :0.0000000   Min.   : 0.0000   Min.   :0.000000   Min.   :0.0000
##  1st Qu.:0.0000000   1st Qu.: 0.0000   1st Qu.:0.000000   1st Qu.:0.0000
##  Median :0.0000000   Median : 0.0000   Median :0.000000   Median :0.0000
##  Mean   :0.0001111   Mean   : 0.2044   Mean   :0.001222   Mean   :0.3957
##  3rd Qu.:0.0000000   3rd Qu.: 0.0000   3rd Qu.:0.000000   3rd Qu.:1.0000
##  Max.   :3.0000000   Max.   :77.0000   Max.   :5.000000   Max.   :1.0000
##
##  num_compromised      root_shell        su_attempted         num_root
##  Min.   :   0.000   Min.   :0.000000   Min.   :0.000000   Min.   :   0.000
##  1st Qu.:   0.000   1st Qu.:0.000000   1st Qu.:0.000000   1st Qu.:   0.000
##  Median :   0.000   Median :0.000000   Median :0.000000   Median :   0.000
##  Mean   :   0.279   Mean   :0.001342   Mean   :0.001103   Mean   :   0.302
##  3rd Qu.:   0.000   3rd Qu.:0.000000   3rd Qu.:0.000000   3rd Qu.:   0.000
##  Max.   :7479.000   Max.   :1.000000   Max.   :2.000000   Max.   :7468.000
```

```
##
##  num_file_creations   num_shells        num_access_files   num_outbound_cmds
##  Min.   : 0.00000   Min.   :0.0000000   Min.   :0.000000   Min.   :0
##  1st Qu.: 0.00000   1st Qu.:0.0000000   1st Qu.:0.000000   1st Qu.:0
##  Median : 0.00000   Median :0.0000000   Median :0.000000   Median :0
##  Mean   : 0.01267   Mean   :0.0004128   Mean   :0.004096   Mean   :0
##  3rd Qu.: 0.00000   3rd Qu.:0.0000000   3rd Qu.:0.000000   3rd Qu.:0
##  Max.   :43.00000   Max.   :2.0000000   Max.   :9.000000   Max.   :0
##
##  is_host_login     is_guest_login         count          srv_count
##  Min.   :0.0e+00   Min.   :0.000000   Min.   :  0.00   Min.   :  0.00
##  1st Qu.:0.0e+00   1st Qu.:0.000000   1st Qu.:  2.00   1st Qu.:  2.00
##  Median :0.0e+00   Median :0.000000   Median : 14.00   Median :  8.00
##  Mean   :7.9e-06   Mean   :0.009423   Mean   : 84.11   Mean   : 27.74
##  3rd Qu.:0.0e+00   3rd Qu.:0.000000   3rd Qu.:143.00   3rd Qu.: 18.00
##  Max.   :1.0e+00   Max.   :1.000000   Max.   :511.00   Max.   :511.00
##
##   serror_rate     srv_serror_rate   rerror_rate   srv_rerror_rate
##  Min.   :0.0000   Min.   :0.0000   Min.   :0.00   Min.   :0.0000
##  1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.00   1st Qu.:0.0000
##  Median :0.0000   Median :0.0000   Median :0.00   Median :0.0000
##  Mean   :0.2845   Mean   :0.2825   Mean   :0.12   Mean   :0.1212
##  3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:0.00   3rd Qu.:0.0000
##  Max.   :1.0000   Max.   :1.0000   Max.   :1.00   Max.   :1.0000
##
##  same_srv_rate    diff_srv_rate     srv_diff_host_rate dst_host_count
##  Min.   :0.0000   Min.   :0.00000   Min.   :0.00000    Min.   :  0.0
##  1st Qu.:0.0900   1st Qu.:0.00000   1st Qu.:0.00000    1st Qu.: 82.0
##  Median :1.0000   Median :0.00000   Median :0.00000    Median :255.0
##  Mean   :0.6609   Mean   :0.06305   Mean   :0.09732    Mean   :182.1
##  3rd Qu.:1.0000   3rd Qu.:0.06000   3rd Qu.:0.00000    3rd Qu.:255.0
##  Max.   :1.0000   Max.   :1.00000   Max.   :1.00000    Max.   :255.0
##
##  dst_host_srv_count dst_host_same_srv_rate dst_host_diff_srv_rate
##  Min.   :  0.0      Min.   :0.0000         Min.   :0.00000
##  1st Qu.: 10.0      1st Qu.:0.0500         1st Qu.:0.00000
##  Median : 63.0      Median :0.5100         Median :0.02000
##  Mean   :115.7      Mean   :0.5212         Mean   :0.08295
##  3rd Qu.:255.0      3rd Qu.:1.0000         3rd Qu.:0.07000
##  Max.   :255.0      Max.   :1.0000         Max.   :1.00000
##
##  dst_host_same_src_port_rate dst_host_srv_diff_host_rate dst_host_serror_rate
##  Min.   :0.0000              Min.   :0.00000             Min.   :0.0000
##  1st Qu.:0.0000              1st Qu.:0.00000             1st Qu.:0.0000
##  Median :0.0000              Median :0.00000             Median :0.0000
##  Mean   :0.1484              Mean   :0.03254             Mean   :0.2845
##  3rd Qu.:0.0600              3rd Qu.:0.02000             3rd Qu.:1.0000
##  Max.   :1.0000              Max.   :1.00000             Max.   :1.0000
##
##  dst_host_srv_serror_rate dst_host_rerror_rate dst_host_srv_rerror_rate
##  Min.   :0.0000           Min.   :0.0000       Min.   :0.0000
##  1st Qu.:0.0000           1st Qu.:0.0000       1st Qu.:0.0000
##  Median :0.0000           Median :0.0000       Median :0.0000
##  Mean   :0.2785           Mean   :0.1188       Mean   :0.1202
```

```
##  3rd Qu.:1.0000           3rd Qu.:0.0000       3rd Qu.:0.0000
##  Max.   :1.0000           Max.   :1.0000       Max.   :1.0000
##
##        class
##  normal   :67342
##  neptune  :41214
##  satan    : 3633
##  ipsweep  : 3599
##  portsweep: 2931
##  smurf    : 2646
##  (Other)  : 4607

##
## Number of records in KDD train set is  125972
```

There are field 'num_outbound_cmds' has only zeros and it should be removed, but I can't do so due to **restriction**, but now I know that I have to take it into account when normalizing data.

Filtering First of all I filtered needed classes, so that I had only 'normal' and 'perl' in result. Also I used `na.exclude()` function to exlude NA's, but it was no NA's in data-set.

Encoding Fields from 2-th to 4-th was encoded with integers by unique criteria. This way grounded for PNN with small Gaussian radius like 0.3, because PNN uses voting power by simillarity of input to patterns (expert knowledge). While for TLP order matters. And it is incorrect to give some order to factors. If there is some logic in protocols order like:

- 1 - UDP, because one of simplests
- 2 - ICMP, because not one of simplest but still one of TCP/IP stack
- 3 - TCP is one of the main protocols of the Internet protocol suite

but they are encoded like this (by alphabetical order):

- 1 - ICMP
- 2 - TCP
- 3 - UDP

so any ordering logic erased and such encoding only can muddle weights of TLP.

But I had to do so and I done so.

Class of signature was to be encoded too, because it is not TLP's purpose to do so. Class 'normal' was encoded with 1 and class 'perl' was encoded with 0 (zero). TLP output interpreted in next way: if output rounds to 0, so it is 'perl', otherwise (if rounded to 1) it is 'normal'.

Normalization Normalization is described by formula:

$$X_{normalized} = \frac{X - \min(X)}{\max(X) - \min(X)}$$

This formula was applied to columns where $\min(X) \neq \max(X)$.

Processed data-set summary After processing NLS KDD, I got 3 .csv files for training NN, validation and testing. But there was an issue: no attacks appeared in validation subset. So I was compelled to insert one record with class 'perl' from training set into validation subset istead of one 'normal' record. The reason why only 1 record consists in next:

```
## KDD train subset has 67343 records of class 'normal' and 3 records of class 'perl'.
## Rate attack/normal is 4.454806e-05
```

```
## KDD validation subset has 13448 records of class 'normal' and 1 records of class 'perl'.
## Rate attack/normal is 7.43605e-05
```

```
## KDD test subset has 9710 records of class 'normal' and 2 records of class 'perl'.
## Rate attack/normal is 0.0002059732
```

It means that data-set has very few attacks to train TLP normally on full data-set and hence to test it normally too.

Data-set supplement I decided to supplement training data-set with repeated records of class 'perl'. I chose supplement rate as 1 attack per 10 normal records. To do so I splitted processed KDD train without attacks into sequences of about 60 elements and added to all of them all 3 records with attack. Then I joined this sequences into single data frame. After that I shuffled got data frame and wrote it into .csv file. I also tried to supplement train set with rates 1/20 and 1/5, but results wasn't very impressive, so I left only this case.

After all data transformations I got 3 new subsets of KDD. All they are look similar. Summary for supplemented train subset:

```
##     duration        protocol_type       service             flag
##  Min.   :0.000000   Min.   :0.0000   Min.   :0.00000   Min.   :0.00000
##  1st Qu.:0.000000   1st Qu.:0.0000   1st Qu.:0.07407   1st Qu.:0.00000
##  Median :0.000000   Median :0.0000   Median :0.07407   Median :0.00000
##  Mean   :0.002884   Mean   :0.1016   Mean   :0.11234   Mean   :0.01023
##  3rd Qu.:0.000000   3rd Qu.:0.0000   3rd Qu.:0.14815   3rd Qu.:0.00000
##  Max.   :0.743861   Max.   :1.0000   Max.   :0.92593   Max.   :0.90000
##    src_bytes           dst_bytes              land          wrong_fragment
##  Min.   :0.0000000   Min.   :0.0000000   Min.   :0.00e+00   Min.   :0
##  1st Qu.:0.0000016   1st Qu.:0.0000158   1st Qu.:0.00e+00   1st Qu.:0
##  Median :0.0000027   Median :0.0000717   Median :0.00e+00   Median :0
##  Mean   :0.0001335   Mean   :0.0005916   Mean   :9.45e-05   Mean   :0
##  3rd Qu.:0.0000035   3rd Qu.:0.0003363   3rd Qu.:0.00e+00   3rd Qu.:0
##  Max.   :1.0000000   Max.   :1.0000000   Max.   :1.00e+00   Max.   :0
##      urgent              hot           num_failed_logins     logged_in
##  Min.   :0.0e+00   Min.   :0.000000   Min.   :0.0000000   Min.   :0.000
##  1st Qu.:0.0e+00   1st Qu.:0.000000   1st Qu.:0.0000000   1st Qu.:0.000
##  Median :0.0e+00   Median :0.000000   Median :0.0000000   Median :1.000
##  Mean   :4.5e-05   Mean   :0.002076   Mean   :0.0003139   Mean   :0.737
##  3rd Qu.:0.0e+00   3rd Qu.:0.000000   3rd Qu.:0.0000000   3rd Qu.:1.000
##  Max.   :1.0e+00   Max.   :0.762376   Max.   :1.0000000   Max.   :1.000
##  num_compromised      root_shell       su_attempted          num_root
##  Min.   :0.00e+00   Min.   :0.00000   Min.   :0.0000000   Min.   :0.00e+00
##  1st Qu.:0.00e+00   1st Qu.:0.00000   1st Qu.:0.0000000   1st Qu.:0.00e+00
##  Median :0.00e+00   Median :0.00000   Median :0.0000000   Median :0.00e+00
##  Mean   :6.16e-05   Mean   :0.09277   Mean   :0.0009315   Mean   :9.29e-05
```

```
##  3rd Qu.:0.00e+00   3rd Qu.:0.00000   3rd Qu.:0.0000000   3rd Qu.:0.00e+00
##  Max.   :1.00e+00   Max.   :1.00000   Max.   :1.0000000   Max.   :1.00e+00
##  num_file_creations   num_shells      num_access_files    num_outbound_cmds
##  Min.   :0.000000   Min.   :0.00000   Min.   :0.0000000   Min.   :0
##  1st Qu.:0.000000   1st Qu.:0.00000   1st Qu.:0.0000000   1st Qu.:0
##  Median :0.000000   Median :0.00000   Median :0.0000000   Median :0
##  Mean   :0.002021   Mean   :0.04574   Mean   :0.0007575   Mean   :0
##  3rd Qu.:0.000000   3rd Qu.:0.00000   3rd Qu.:0.0000000   3rd Qu.:0
##  Max.   :0.430000   Max.   :1.00000   Max.   :1.0000000   Max.   :0
##  is_host_login      is_guest_login        count            srv_count
##  Min.   :0.00e+00   Min.   :0.00000   Min.   :0.000000   Min.   :0.000000
##  1st Qu.:0.00e+00   1st Qu.:0.00000   1st Qu.:0.001957   1st Qu.:0.001957
##  Median :0.00e+00   Median :0.00000   Median :0.005871   Median :0.007828
##  Mean   :1.35e-05   Mean   :0.01178   Mean   :0.040238   Mean   :0.049431
##  3rd Qu.:0.00e+00   3rd Qu.:0.00000   3rd Qu.:0.023483   3rd Qu.:0.031311
##  Max.   :1.00e+00   Max.   :1.00000   Max.   :1.000000   Max.   :1.000000
##   serror_rate      srv_serror_rate    rerror_rate      srv_rerror_rate
##  Min.   :0.00000   Min.   :0.00000   Min.   :0.00000   Min.   :0.00000
##  1st Qu.:0.00000   1st Qu.:0.00000   1st Qu.:0.00000   1st Qu.:0.00000
##  Median :0.00000   Median :0.00000   Median :0.00000   Median :0.00000
##  Mean   :0.01222   Mean   :0.01098   Mean   :0.04018   Mean   :0.04057
##  3rd Qu.:0.00000   3rd Qu.:0.00000   3rd Qu.:0.00000   3rd Qu.:0.00000
##  Max.   :1.00000   Max.   :1.00000   Max.   :1.00000   Max.   :1.00000
##  same_srv_rate    diff_srv_rate     srv_diff_host_rate dst_host_count
##  Min.   :0.0000   Min.   :0.00000   Min.   :0.0000     Min.   :0.0000
##  1st Qu.:1.0000   1st Qu.:0.00000   1st Qu.:0.0000     1st Qu.:0.1843
##  Median :1.0000   Median :0.00000   Median :0.0000     Median :0.6549
##  Mean   :0.9721   Mean   :0.02617   Mean   :0.1148     Mean   :0.5944
##  3rd Qu.:1.0000   3rd Qu.:0.00000   3rd Qu.:0.0800     3rd Qu.:1.0000
##  Max.   :1.0000   Max.   :1.00000   Max.   :1.0000     Max.   :1.0000
##  dst_host_srv_count dst_host_same_srv_rate dst_host_diff_srv_rate
##  Min.   :0.0000     Min.   :0.0000         Min.   :0.00000
##  1st Qu.:0.2471     1st Qu.:0.4600         1st Qu.:0.00000
##  Median :0.9922     Median :1.0000         Median :0.00000
##  Mean   :0.6790     Mean   :0.7393         Mean   :0.03921
##  3rd Qu.:1.0000     3rd Qu.:1.0000         3rd Qu.:0.02000
##  Max.   :1.0000     Max.   :1.0000         Max.   :1.00000
##  dst_host_same_src_port_rate dst_host_srv_diff_host_rate dst_host_serror_rate
##  Min.   :0.000               Min.   :0.00000             Min.   :0.00000
##  1st Qu.:0.000               1st Qu.:0.00000             1st Qu.:0.00000
##  Median :0.010               Median :0.00000             Median :0.00000
##  Mean   :0.111               Mean   :0.02363             Mean   :0.01266
##  3rd Qu.:0.060               3rd Qu.:0.03000             3rd Qu.:0.00000
##  Max.   :1.000               Max.   :1.00000             Max.   :1.00000
##  dst_host_srv_serror_rate dst_host_rerror_rate dst_host_srv_rerror_rate
##  Min.   :0.00000          Min.   :0.00000      Min.   :0.00000
##  1st Qu.:0.00000          1st Qu.:0.00000      1st Qu.:0.00000
##  Median :0.00000          Median :0.00000      Median :0.00000
##  Mean   :0.00556          Mean   :0.06326      Mean   :0.04063
##  3rd Qu.:0.00000          3rd Qu.:0.00000      3rd Qu.:0.00000
##  Max.   :1.00000          Max.   :1.00000      Max.   :1.00000
##      class
##  Min.   :0.0000
##  1st Qu.:1.0000
```

```
##  Median :1.0000
##  Mean   :0.9091
##  3rd Qu.:1.0000
##  Max.   :1.0000

##
## Number of records in KDD train set is  74078

## KDD validation subset has 67343 records of class 'normal' and 6735 records of class 'perl'.
## Rate attack/normal is 0.1000104
```

As we can see, 'attack/normal' rate is about 0.1 now and data-set got fat. It might appear a question why fields 'duration', 'service', 'flag', 'hot' and 'num_file_creations' has maximum value less than 1. It is because it's just a part of full data-set. Normalization was applied to glued train, validation and test subsets. So maximums for these rows left in test subset. I checked summary for it and it really is.

Computations

I trained two-layer prezeptron and executed recognition more than 10 times, but save results only for 3 cases. It takes a lot of time but I was persistent and tried many times to get expected result.

At first time I trained TLP on just processed data set.
At second time I trained it on supplemented (with rate 1/10) train set.
At third time I trained it on same train set for 3000 epochs instead of 100.

It's obviously that at third time I done the same but with more calculations than in second time.

PNN solution

Before showing results of training TLP I will tell about provided PNN for this task to compare results. I byilt PNN from laboratory work 2 on 3000 records from processed train set (it was records from 35000 to 36999 and from 54000 to 54999). These records include all 3 attack records.

PNN implemented on different classes.

Neurons

`class Neuron` implements classical neuron. It requires to specify number of incoming synaptic links `inputs_num` and allows to specify transfer function, activation function, bias (`shift`) and weights (set randomly by default). It has next methods:

- `get_inputs_num(self)` - simple getter for `inputs_num`
- `_set_random_weights(self, bipolar=True)` - protected method dedicated to set weights randomly on initialization
- `feed(self, X)` - returns neurons activation value (out signal) for incoming signal `X`. `X` expected to be one-dimensional list. It does not compute outcome itself, but calls `_feed`
- `_feed(self, X)` - protected method that implements calculations for `feed` method

`class InputNeuron(Neuron)` is dedicated for input neuron, and it can be initialized without any provided parameters. It has linear activation function and skips use of transfer functions because it has single input (equivalent of sensitive cell). Also it can have name for input variable and has according getter and setter. As mentioned it's `feed` method doesn't use transfer function.

`class PatternNeuron(Neuron)` corresponds to a neuron in patten layer. It's necessary to specify weights (values list of training example) to initialize it. Also it uses such combination of transfer function and activation function that it is the Gaussian function ultimately. Gaussian function:

$$G(X) = \sum \exp(\frac{-(W-X)^2}{\sigma^2})$$

where σ – gaussian radius, X – predictor vector, W – weights.

I used next transfer and activation:

$$Transfer(X) = W - X$$

$$Activate(X) = \sum \exp(\frac{-(X)^2}{\sigma^2})$$

So composition is:

$$Activate(Transfer(X)) = G(X)$$

`class SummaNeuron(Neuron)` corresponds to neuron in summation layer. It initialized with number of incoming synaptic links as parameter `inputs_num`. It has summa transfer function and linear activation function with coefficient $\frac{1}{inputs\_num}$.

`class OutputNeuron(Neuron)` corresponds to output neuron. It initialized with number of inputs equal to number of classes. It has 'whichmax' transfer function, which returns number of element with max value in inputs. Activation function just returns name of class according to transfered number. This neuron can be initialized with set of classes names, but by default it returns number of class with max vote. Also it has setter `set_class_names(self, class_names)` for `_class_names` attribute.

Layers

`class NeuronsLayer` implements interface to bind several neurons into single layer to work with all of them through single object. `NeuronsLayer` initialized with list of objects of class `Neuron`. It has just one public method `feed(self, X, overall_input=False)`, which runs through all neurons in layer and calls their `feed` method. It feeds to them same input `X` in case parameter `overall_inputs` set as True, otherwise it expects that `X` is two-dimensional list and feed rows to each neuron.
To use `overall_inputs = True` each neuron in layer must have same `inputs_num`.

Probability neural network

`class PNN` initialized with two-dimensional array `train_in` of predictors in each row and list `train_out` of classes corresponding to predictors. It is available to mention names of input variables `input_names` and gaussian radius `gaussian_radius` (0.3 by default) while initialization.

It has single public method `recognize(self, X)` which juxtaposes classes to predictors `X`. If `X` is one-dimensional, than it returns single class name, else if `X` is two-dimensionsl, than it expects predictors in rows and returns list of recognized class names.

Results analysis

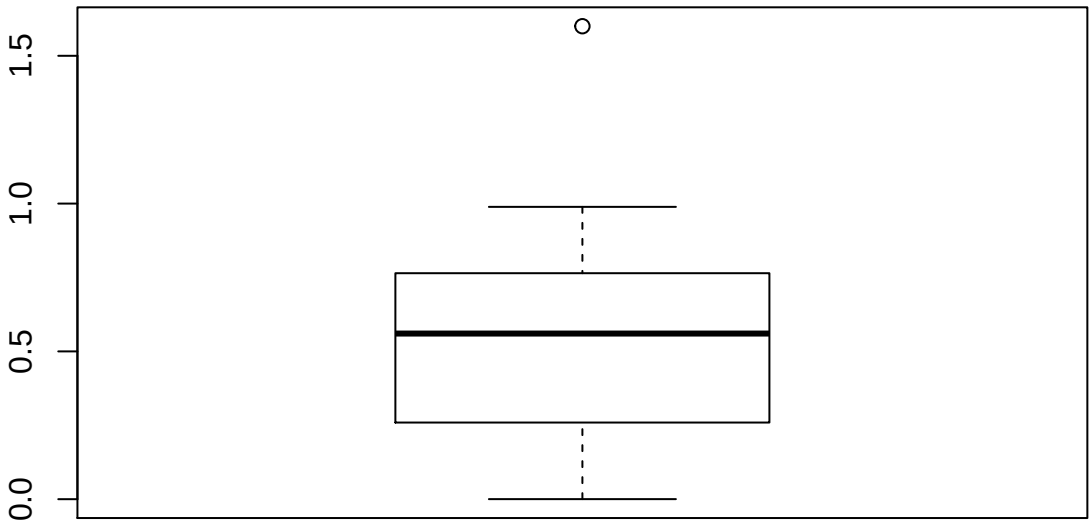
It's not convenient to see full recognition of validation and testing sets at least because it takes a lot of space and time to consider, so I will use boxplots. Boxplots are very informative to view errors of TLP. It's well interpreted and reliable due to just one attack in validation set and pair attacks in test set.

Boxplots interpretation rules

Error was calculated with formula $err = Y - Y_{web}$, where Y - expected, Y_{web} - TLP output.

- If **expected 'normal'** (encoded as 1) and NN **recognized it**, so it's output is between 0.5 and 1, hence $err \in [0, 0.5]$
- If **expected 'normal'** (encoded as 1) and NN **did not recognize it**, so it's output is between 0 and 0.5, hence $err \in [0.5, 1]$
- If **expected 'perl'** (encoded as 0) and NN **recognized it**, so it's output is between 0 and 0.5, hence $err \in [-0.5, 0]$
- If **expected 'perl'** (encoded as 0) and NN **did not recognize it**, so it's output is between 0.5 and 1, hence $err \in [-1, -0.5]$

Also remind description of boxplot on example:



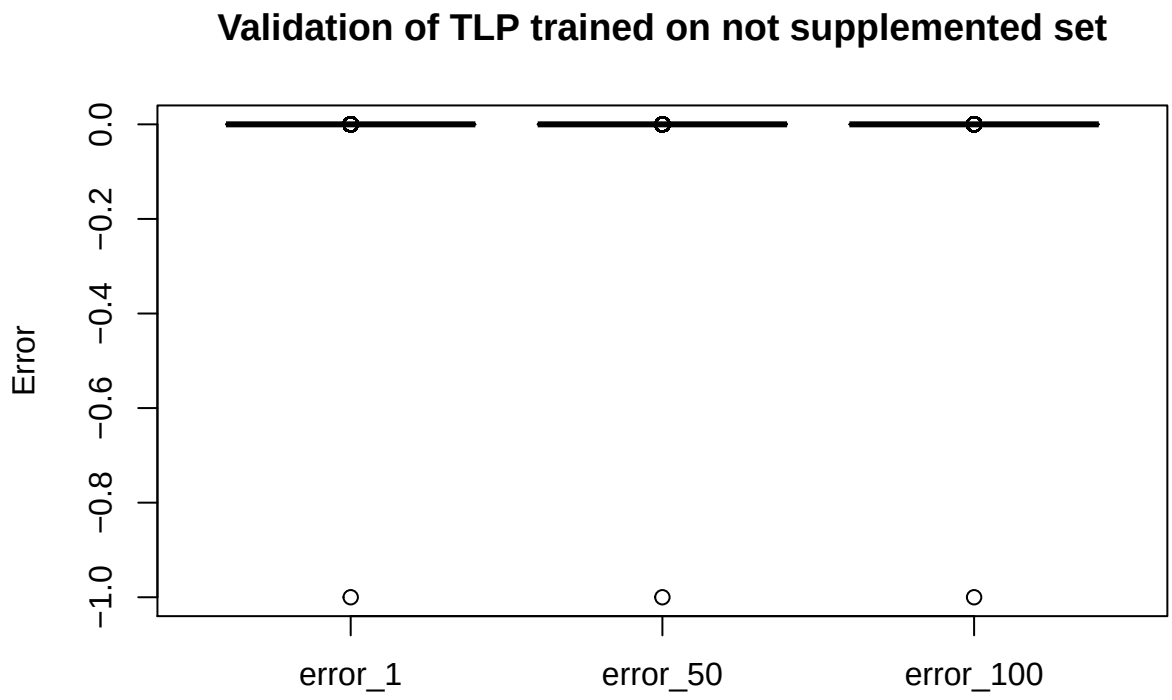
- Bold line inside of box is median (second quartile)
- Box corresponds to interquartile range, that means lower bound of box is first quartile, and upper one is third quartile
- Whiskers shows the farthest observation, wich is not farther than 1.5 times interquartile range from the box
- Points shown farther than whiskers end and often recognized as ejections (but not in our case)

So less words and more plots.

Error plots for TLP trained on not supplemented train set

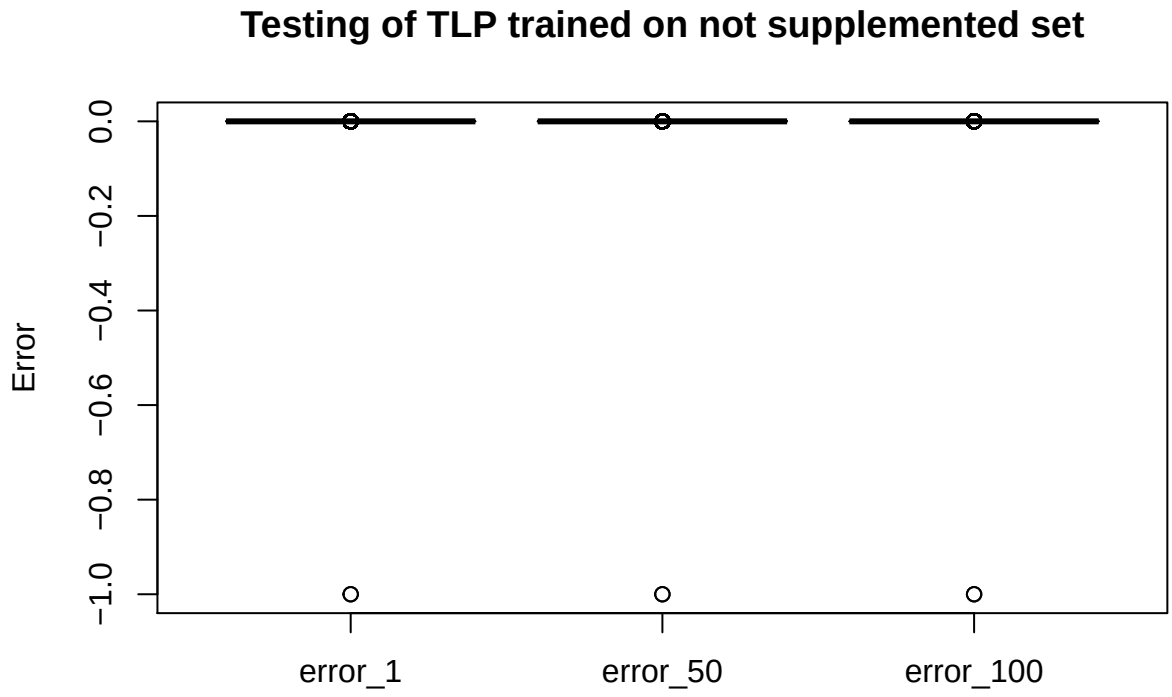
Validation

`error_1`, `error_50` and `error_100` are respective to errors after 1-th, 50-th and 100-th train epochs.



Using mentioned **boxplots interpretation rules**, we can say that all examples in this case was recognized as 'normal' and hence single 'perl' wasn't recognized. Bad result. And also there is no outputs significally different from 1. TLP failed validation.

Testing



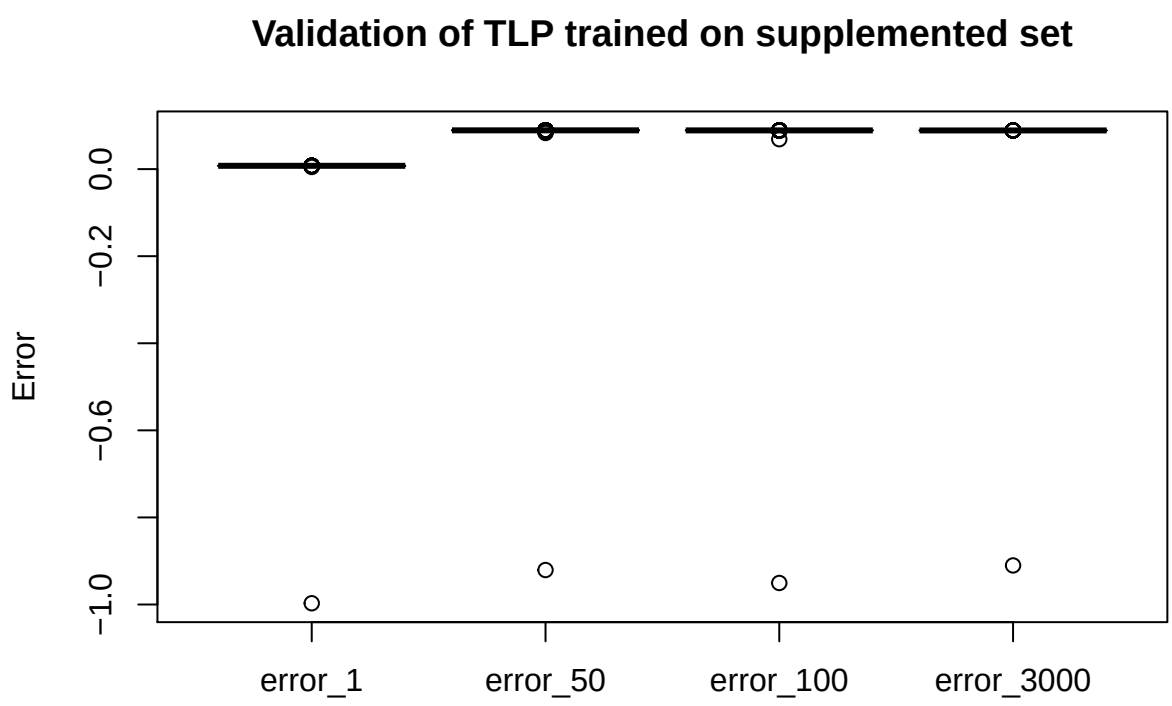

```
## Number of recognized attacks
## after epoch 1: 0/2
## after epoch 50: 0/2
## after epoch 100: 0/2
```

Nothing new. It doesn't work at all.

Error plots for TLP trained on supplemented train set

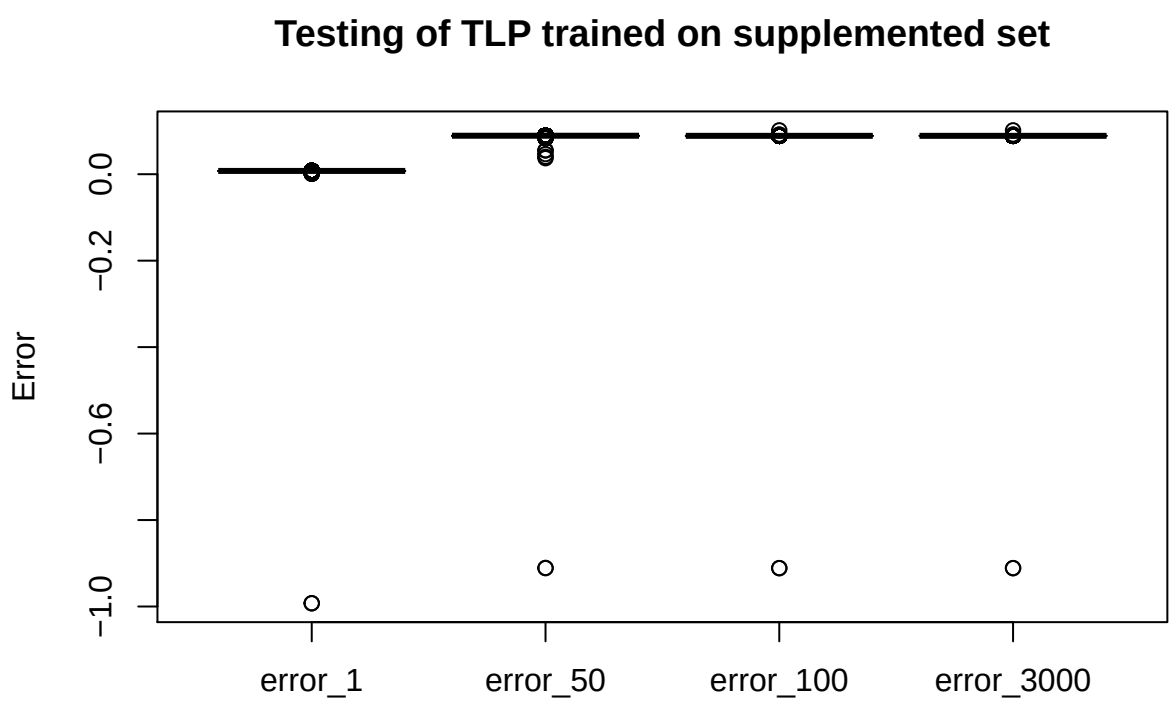
As I said, I tried to train TLP also for 3000 epochs. This case wasn't essential but I tried it and it will be reviewed.

Validation



There are some fluctuations, but NN still doesn't work correctly. Yeah! it recognizes almost all set! But it does not recognize attack. In such way this NN is not more than trash. Validation failed. And there is no expectations of succes testing.

Testing




```
## Number of recognized attacks
## after epoch 1: 0/2
## after epoch 50: 0/2
## after epoch 100: 0/2
## after epoch 3000: 0/2
```

It looks more interesting. Recognition changed. But attacks still not recognized. I suggested that it can be recognized, but outputs for recognized attack not in range from 0 to 0.5, but in it significantly less than outputs for normals. So let's ckeck it.

```
##   recognized_1       recognized_50       recognized_100
##  Min.   :0.9924609   Min.   :0.9109803   Min.   :0.9109377
##  1st Qu.:0.9924609   1st Qu.:0.9109973   1st Qu.:0.9111024
##  Median :0.9924609   Median :0.9109973   Median :0.9111024
##  Mean   :0.9924662   Mean   :0.9110258   Mean   :0.9111433
##  3rd Qu.:0.9924609   3rd Qu.:0.9109973   3rd Qu.:0.9111024
##  Max.   :0.9991516   Max.   :0.9630123   Max.   :0.9672081
```

```
##      recognized_1 recognized_50 recognized_100
## 1787    0.9924609     0.9109973      0.9111024
## 9511    0.9924609     0.9109973      0.9111024
```

Everything is clear there. Values of outputs when attack fall into interquartile range and fourth quartile. That means NN recognized attacks like common normals. If these values were very close to min values, than

it could be encouraging, but no. This purpose to learn TLP on three examples of attacks and 67 thouthands of normals to recognize attack is desperate.

Last one thing that I should check is how PNN cope with this task.

PNN recognition results analysis

As I mentioned, PNN was trained on 3 thouthands of examples and it was recognizing relatively fast, faster than TLP trained. Results of recognition won't be plotted because output is class.

I will use F1-score and accuracy (in percents) as measures of a test's accuracy:

$$F_1 = 2\frac{precision \times recall}{precision + recall}$$

$$ACC = \frac{TP + TN}{TP + TN + FN + FP}$$

where $precision = \frac{TP}{TP+TN}$, $recall = \frac{TP}{TP+FN}$,
TP - number of true-positives, TN - number of true-negatives,
FN - number of false-negatives, FP - numer of false-positives.

I decided to choose class 'perl' as positive, because purpose of NN is to detect attacks.

Validation

```
## Attacks recognized (TP) - 12513
## Attacks not recognized (FP) - 935
## Normals recognized (TN) - 1
## Normals not recognized (FN) - 0
##
## Accuracy: ACC = 93.05%
## F1 = 0.99996
```

Very high accuracy (for 3K examples of 67K) and F1-score almost 1, which is very good

Testing

```
## Attacks recognized (TP) - 12513
## Attacks not recognized (FP) - 935
## Normals recognized (TN) - 1
## Normals not recognized (FN) - 0
##
## Accuracy: ACC = 93.05%
## F1 = 0.99996
```

Also very high accuracy and F1-score almost 1.

These results prove that TLP is not relevant for recognition, when number of attack records is very small in rate to normal ones unlike PNN.

Conclusion

I haven't reached succes in training two-layer perzeptron on NLS KDD data-set to recognize cyber attacks of type 'perl'. I associate it with deficiency of training examples of 'perl' cyber attacks in training set. But I believe that this problen solvable, that is to say it exists set of weights wich allows TLP to recognize 'perl' cyber attacks from NLS KDD, even if classes not linearly separable. Theory says that TLPs can solve such problems. But question in how complicated way to reach this. And it is definately complicated, because trainings take a lot of time. This amount of time not relevant if there is no need to recognize very big number of data at little time. PNN is much more effective and also easier to build. PNN also much more relevant, due to it's single usage. Also PNN can be easily set up, it has single type of parameter - radius of Gaussian function. It can be chosen expertly for each pattern neuron, but even default value 0.3 works good on normalized data set. And PNN is interpretable unlike TLP for classification problem. But PNN has some problems, like generalizability and it slows down with big amount of data and takes a lot of memory. Slowing down is critical because TLP have to be trained once, probably it takes a lot of time, but still TLP recognizes much faster, than PNN with wide pattern layer.