FINAL

Neural Network Intrusion Detection System for MQTT-based IoT Network

Abstract

We explore the application of neural network in cyber security in the field of IoT. Specifically, we apply neural networks to build an intrusion detection system IDS by using a recently published dataset in IEEE. The data set is obtained from a benchmark MQTT protocol-based sensors network.

Jawaher Albanki

20143551

Table of Contents

Objective 2

Introduction 2

Procedure and Design 5

Results 6

Python 6

Bidirectional Flow 7

Unidirectional Flow 7

MATLAB 8

Discussion 12

References 14

Appendix A 15

Appendix B 23

Objective

The objective of this project is to build an artificial neural network Intrusion Detection System for MQTT-based IoT network.

Introduction

Security in the of internet of things is currently having a growing research attention. IoT devices are battery and resource constraint. As such, they only use resource constrained protocols to achieve their communication. Among the resource constrained communication protocols used for IoT are CoAP, MQTT, and others. Current best security practices and protocols such as TLS are usually hard to integrate with such protocols, as they tend to require advanced processing which would render such protocols are no longer resource constrained. Therefore, IoT traffic tends to be unencrypted. This makes a network vulnerable to eavesdrop and intrusion. As such there have been massive efforts to implement strong intrusion detection systems in an IoT environment to detect hackers and their malicious activities. Identifying such activities is quite a challenging task. An algorithm is required to learn and understand a pattern to determine if its benign or malicious.

Currently, research is focused towards introducing advanced artificial intelligence and machine learning technologies in the security of IoT to help detect malicious activities in a network. Such technologies are implemented in the form of an intrusion detection system monitoring a network environment that has IoT or other devices. An IDS is supposed to identify hackers in a system to give firewalls a signal to block their activity.

One of the major problems facing researchers in developing an artificially intelligent IDS for IoT systems is the scarcity of dataset to train their algorithms. Hanan et al. in their paper contributed to the field by making a large dataset available in IEEE Dataset website for training of artificially intelligent IDS. The authors' dataset is specifically extracted from a network environment in which there are IoT sensors and a camera that communicate with each other through MQTT protocol. As such, this dataset is to specifically train intelligent IDS to detect attacks in an MQTT environment. They tested multiple machines learning classification algorithm. They are the following:

- Logistic Regression
- k-Nearest Neighbors
- Gaussian Naive Bayes
- Decision Trees
- Random Forests
- Support Vector Machine (linear and RBF kernel)

The data that were used to do the training and testing are features extracted from PCAP network packet scanning software. The extracted packet features are classified in one of three types:

- Packet-based features
- Unidirectional flow-based features
- Bidirectional flow-based features

The only difference between them is the features which they use as inputs. The way they are organized are shown in next table. Packet based only considers features that are packets. Unidirectional based only considers features that are relating to the flow data. Bidirectional based is same as unidirectional except

Tables taken from [1]

Feature	Data Type	Description	Packet	Uni-flow	Bi-flow
ip_src	Text	Source IP Address	✓	✓	✓
ip_dest	Text	Destination IP Address	✓	✓	✓
protocol	Text	Last layer protocol	✓		
ttl	Integer	Time to live	✓		
ip_len	Integer	Packet Length	✓		
ip_flag_df	Binary	Don't fragment IP flag	✓		
ip_flag_mf	Binary	More fragments IP flag	✓		
ip_flag_rb	Binary	Reserved IP flag	✓		
prt_src	Integer	Source Port	✓	✓	✓
prt_dst	Integer	Destination Port	✓	✓	✓
proto	Integer	Transport Layer protocol (TCP/UDP)		✓	✓
tcp_flag_res	Binary	Reserved TCP flag	✓		
tcp_flag_ns	Binary	Nonce sum TCP flag	✓		
tcp_flag_cwr	Binary	Congestion Window Reduced TCP flag	✓		
tcp_flag_ecn	Binary	ECN Echo TCP flag	✓		

Table 1 continued

Feature	Data Type	Description	Packet	Uni-flow	Bi-flow
tcp_flag_urg	Binary	Urgent TCP flag	✓		
tcp_flag_ack	Binary	Acknowledgement TCP flag	✓		
tcp_flag_push	Binary	Push TCP flag	✓		
tcp_flag_reset	Binary	Reset TCP flag	✓		
tcp_flag_syn	Binary	Synchronization TCP flag	✓		
tcp_flag_fin	Binary	Finish TCP flag	✓		
num_pkts	Integer	Number of Packets in the flow		✓	*
mean_iat	Decimal	Average inter arrival time		✓	*
std_iat	Decimal	Standard deviation of inter arrival time		✓	*
min_iat	Decimal	Minimum inter arrival time		✓	*
max_iat	Decimal	Maximum inter arrival time		✓	*
num_bytes	Integer	Number of bytes		✓	*
num_psh_flags	Integer	Number of push flag		✓	*
num_rst_flags	Integer	Number of reset flag		✓	*
num_urg_flags	Integer	Number of urgent flag		✓	*
mean_pkt_len	Decimal	Average packet length		✓	*
std_pkt_len	Decimal	Standard deviation packet length		✓	*
min_pkt_len	Decimal	Minimum packet length		✓	*
max_pkt_len	Decimal	Maximum packet length		✓	*
mqtt_messagetype	Integer	MQTT message type	✓		
mqtt_messagelength	Binary	MQTT message length	✓		
mqtt_flag_uname	Binary	User Name MQTT Flag	✓		
mqtt_flag_passwd	Binary	Password MQTT flag	✓		
mqtt_flag_retain	Binary	Will retain MQTT flag	✓		
mqtt_flag_qos	Integer	Will QoS MQTT flag	✓		
mqtt_flag_willflag	Binary	Will flag MQTT flag	✓		
mqtt_flag_clean	Binary	Clean MQTT flag	✓		
mqtt_flag_reserved	Binary	Reserved MQTT flag	✓		
is_attack	Binary	1 if the instance represents an attack, 0 otherwise.	x	x	x

* represented as two features in the biflow features file (forward fwd and backward bwd)

in some cases, it has flow in both directions represented. Each type-based feature is in a separate excel sheet and is treated exclusively. Each excel sheet contains no less than 30000 data extracted. In addition to having the features, an output is dedicated to the single output of the IDS which can be thought of as a binary number (0 or 1) which indicates if the packet is an attack or not.

Due to technical difficulties with the code, we limited the scope of our work only to unidirectional flow data and bidirectional flow data.

Each dataset from the above consists of four types of packets. One normal packet and the other four are attacks which are:

- Aggressive Scan (Scan A) – Done by N-map software used to simulate network traffic
- User Datagram Protocol UDP scan (Scan SU) – done by
- Sparta SSH brute-force (Sparta) – Sparta is a penetration testing software used in this attack
- MQRR brute-force attack (MQTT_BF)

In this project, we continue on the work of Hanen and use their dataset to build an artificial neural network-based IDS. We will use backpropagation technique to make the training process more efficient. We will then visualize our results and finally cross validate.

The dataset had the setup displayed in the figure below. Paper [1] describes it in further detail. There are 12 sensors in the network, one camera, one hacker, an MQTT broker server, and a camera.

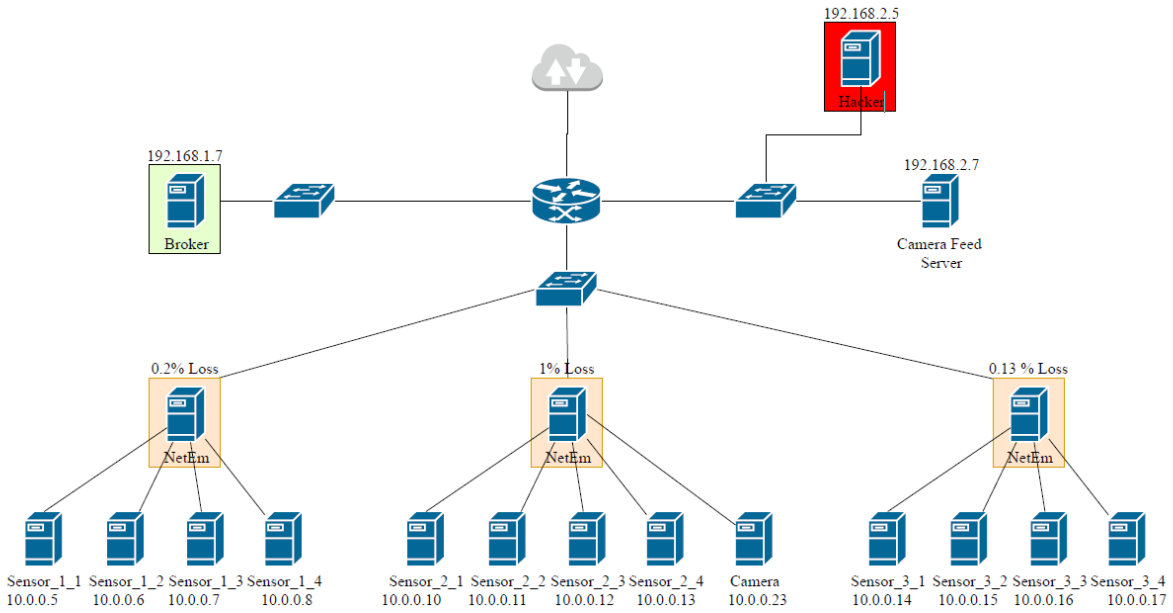


Figure taken from [1]

Procedure and Design

Two codes will be written to train two different neural networks. The first code is a python code. The second approach is using MATLAB neural network application. MATLAB was used for the sakes of trying.

The code is actually taken from GitHub. It is published by the authors of paper [1] along with their data set. The code does automatically almost everything required by this assignment question. The way it works is it first extracts the x and y data from the dataset which is a big data set containing around 31000 columns of data. Then, it shuffles the data and divides part of it for training and the other part for testing. Then it initializes the classification method to call the classification instance and start the training and testing respectively. Finally, the code prints all the outputs into '.csv' folders.

One more important thing to mention about the code is that the input features dataset of x are strings not a quantity. The relation of strings rather than numbers to an output is quite a challenge. What needs to happen is we need to encode those strings and after training the neural network we decode them back again if we need do. The encoding process is also included within the code that we use. It automatically encodes the data before assigning them to x. That is why, when we use MATLAB as we will see later, the inputs are already encoded.

Despite the code being complete and consistent of almost everything we need, it does not have an neural network classifier. The only classifiers it has are the ones mentioned in the introduction used by the original authors. Therefore, the code has been modified by removing the machine learning algorithms used by the authors because they are not needed anymore and writing the part that will perform the neural network. The added part is as shown below.

```
# 0- Nueral Net
ann_classifier = MLPClassifier(solver='sgd', alpha=1e-5, hidden_layer_sizes=(8,8,7, 2), random_state=1)
classify_sub(ann_classifier,
             x_train, y_train,
             x_test, y_test,
             confusion_matrix_folder + prefix + '_cm_ANN.csv',
             summary_folder + prefix + '_summary_ann.csv',
             'ANN',
             verbose)
```

The above code kept diverging. For this reason, there is one little modification that has been done. The alpha constant was raised to 10. This made the neural network converge with very good results. However, the running time was very slow. It took around more than 45 minutes to run.

The code basically constructs a neural network using the 'sklearn' library in python. The neural network is a classification type neural network. The solver that is selected is 'sgd' which stands for stochastic gradient decent. This algorithm optimizes the neural network parameters until it reaches optimal hyperparameters. The hidden layers are four layers with 8, 8, 7, and 2 activation functions respectively. The activation function is kept as default which is a 'relu' activation function.

The 'classify_sub' function is a function written by the authors of paper [1] and code developers to perform the classification and train the selected algorithm. The results of the python code are displayed in the results section.

The code in appendix A was run to display the results in the results section. The outputs that is display are:

- Accuracy matrix
- Confusion matrix
- Cross Validation

The accuracy matrix is the main result. It consists of: Recall, Precision, and F1-Score. They are calculated as follows:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2TP}{2TP + FP + FN}$$

Where TP is true positive, FP is false positive, FN is false negative. The results also show 'accuracy' which is calculated as follows

$$Overall\ Accuracy = \frac{TP + TN}{P + N}$$

Where TP is true positive, TN is true negative, P is positive, and N is negative.

It performs multiple iterations to train the neural network. In each iteration

The second approach to solve this problem was to prepare the x and y data in '.csv' folders. That was done using the same code of the authors of [1]. However, the classifications and their algorithms are entirely omitted. Then, the x and y data were trained via MATLAB neural network application.

Results

Running the code gave the following results

Python

The python code of the others was modifying and used for our results. However, the code seemed to have difficulties in processing the "Packet" data. As such, only bidirectional flow and unidirectional flow datasets were used to test the neural network.

The data sets that we used we shuffled and split into two segments. One segment is kept for training which is the larger one, representing 75% of all data. The other segment is kept for testing, which is the smaller one, representing 25% of all data. The part of the code that does this operation is the following

```
x_train, x_test, y_train, y_test = train_test_split(x, y,
                                   test_size = 0.25,
                                   random_state = 42)
```

Bidirectional Flow

	precision	recall	f1-score	support
Benign	0.974475	0.999418	0.986789	1719
Scan A	0.991957	0.922693	0.956072	401
Scan SU	1	0.988789	0.994363	446
Sparta	1	1	1	2823
MQTT_BF	0.995859	0.992094	0.993973	2909
accuracy				0.992769
macro avg	0.992458	0.980599	0.986239	8298
weighted avg	0.992872	0.992769	0.992725	8298

Confusion matrix

	Benign	Scan A	Scan SU	Sparta	MQTT_BF
Benign	3436	0	0	0	0
Scan A	26	684	87	0	0
Scan SU	101	31	314	0	0
Sparta	0	0	0	5647	0
MQTT_BF	40	72	0	1	5662

Unidirectional Flow

	precision	recall	f1-score	support
Benign	0.928417	1	0.96288	3437
Scan A	0.969125	0.708908	0.818841	797
Scan SU	0.88668	1	0.939937	446
Sparta	1	1	1	5647
MQTT_BF	1	0.981295	0.990559	5774
accuracy				0.978883
macro avg	0.956844	0.938041	0.942443	16101
weighted avg	0.980052	0.978883	0.97806	16101

Confusion matrix

	Benign	Scan A	Scan SU	Sparta	MQTT_BF
Benign	3436	0	0	0	0
Scan A	26	684	87	0	0
Scan SU	101	31	314	0	0
Sparta	0	0	0	5647	0
MQTT_BF	40	72	0	1	5662

There are two common methods of validating the neural network to reach hyperparameters:

- **Cross-validation method (K-folds)**
- **Holdout validation method (Dropout Rate)**

In the above python code, the method that was used is cross-validation (k-fold) with 5 number of folds. Nevertheless, it is also worth mentioning that the k-fold method involves repeating the process of training the network for the number of k-fold times specified, which is in our case equal to five. At the end, the average of all data is taken as our result. In the next code, where we use MATLAB instead, the method that will be used for validation is holdout validation method.

MATLAB

MATLAB has better and easier ways of visualizing data. This can make the process of trial and error easier. As part of the project, MATLAB as well as python was tested out. However, the data preparation and encoding are the only part which were left to Python. Again, the original authors code was modified such that the code returns the data that we need in .csv files that will be used in MATLAB, in the next stage. This was done by using the main function and adding the following lines of code before deleting the training and testing variables.

```
np.savetxt('x_train.csv',x_train,delimiter=',')
np.savetxt('x_test.csv', x_test, delimiter=',')
np.savetxt('y_train.csv', y_train, delimiter=',')
np.savetxt('y_test.csv', y_test, delimiter=',')
```

The above line produces four .csv folders however, in MATLAB, since there is already a functionality to specify the percentage of training and testing data, and the percentage of validation data in addition to that, there is not needed to have a separate folder for training and testing. We combined ‘x’ folders together and ‘y’ folders together, as a single matrix. This was done in MATLAB command window in the same directory of the .csv folder.

```
>> y1=load('y_test.csv');
>> y2=load('y_train.csv');
>> y=[y1;y2];
>> y1=load('y_test.csv');
>> y2=load('y_train.csv');
>> y=[y1;y2];
>> z=[x y];
```

The two above steps can be applied to both bidirectional flow and unidirectional flow. However, we will limit MATLAB to bidirectional flow. As the main intention is to test MATLAB’s performance on neural networks.

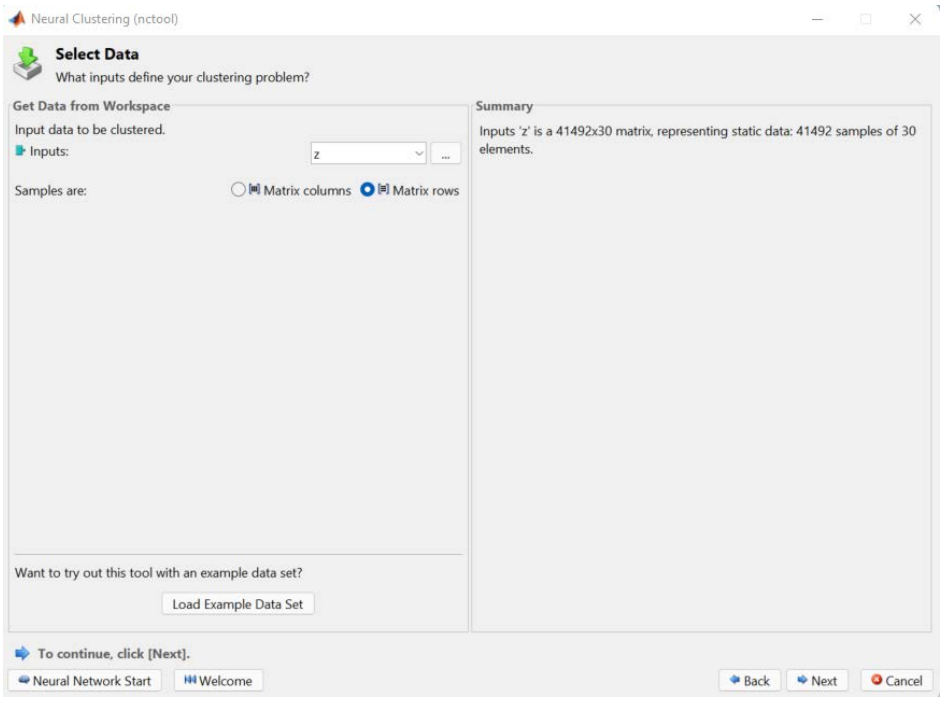


Figure – clustering neural network application

The number of sizes of two-dimensional map was set to 100 after a number of trail and errors. The number of epochs was made equal to 100 epochs. Each epoch by default in MATLAB has a number of 200 iterations.

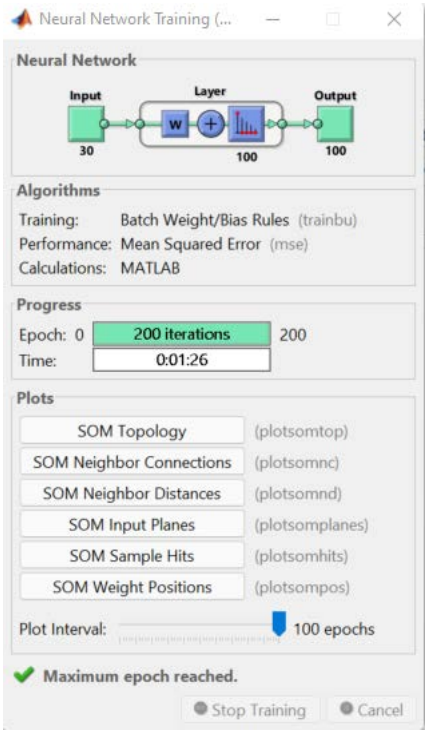


Figure – training window

The application that was selected is the pattern recognition neural network because it can also be used for classification problems which is what we are dealing with in our case.

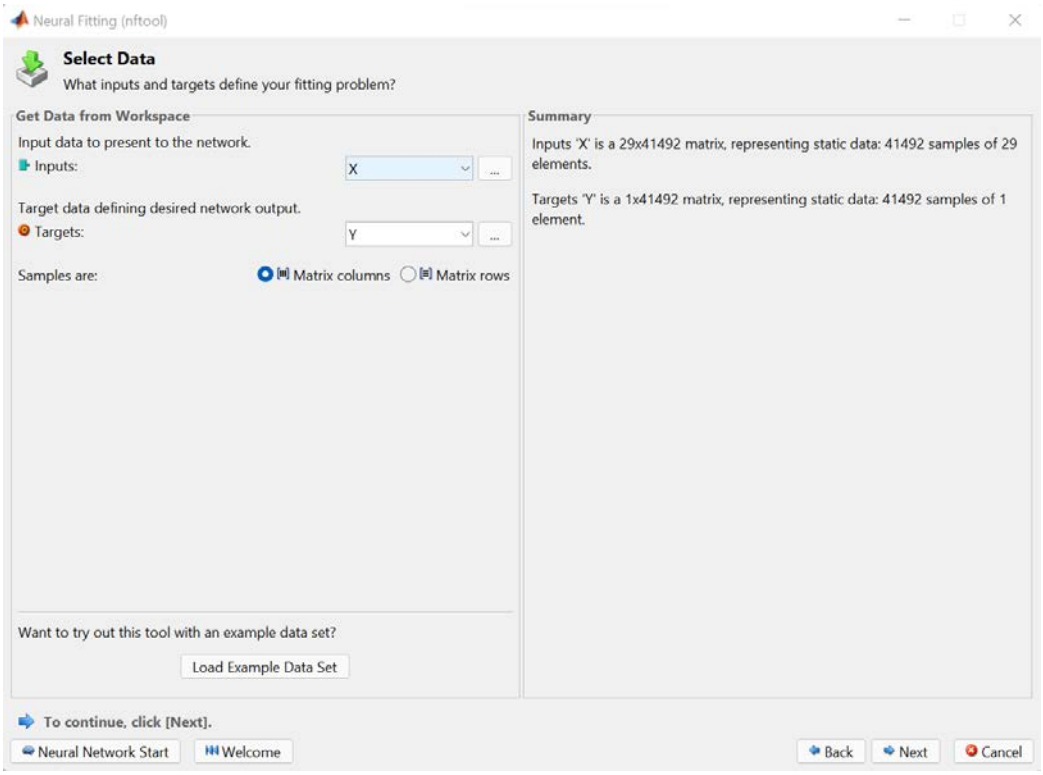


Figure – selecting the datasets in neural fitting application

Then, we partitioned the data into training, validation, and testing data. Notice that before in the python code we selected 75% training, 25% testing, and as for the validation we used cross-validation by k-fold method. Here, the case is different because the validation method is different. That is why we allocated 75% for testing, 15% for validation, and 10% for testing.

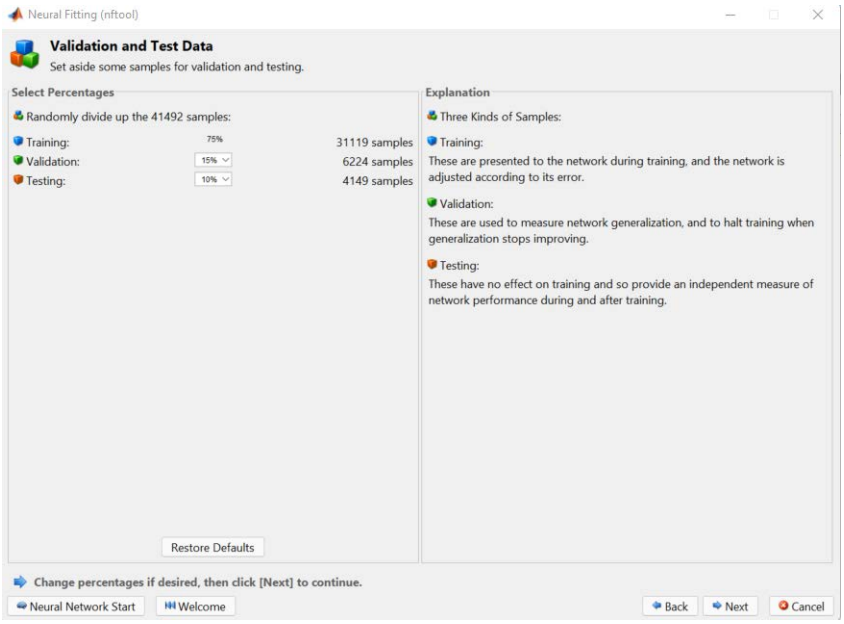


Figure – allocating training, testing, and validation data percentages

The number of hidden neurons we kept as 30. It is an accepted rule of thumb to set the number of hidden neutrals equal to the size of input plus output. Nevertheless, this still is a trial-and-error procedure.

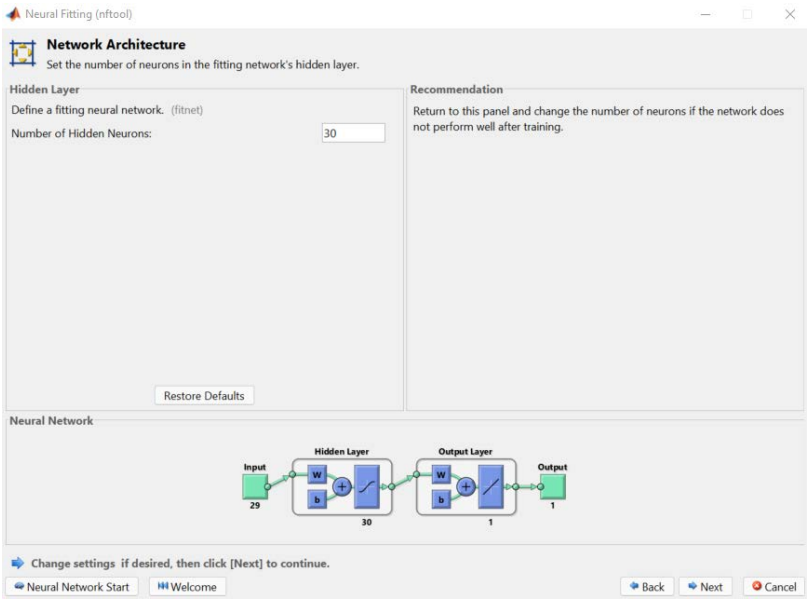


Figure – selecting number of hidden neurons

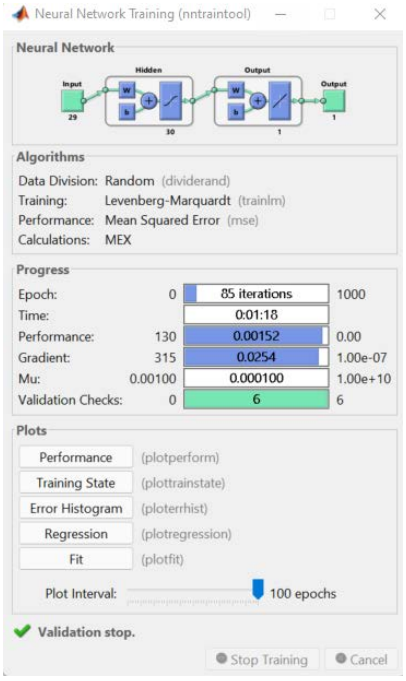


Figure – the neural network training window

The training optimization algorithm was selected as Levenberg-Marquardt which takes less time but more memory. We needed a fast algorithm regardless of memory which is why we selected this algorithm.

The results for 30 neurons are shown in Appendix B. However, the accuracy which can be seen from the performance plot is not as desired. The curve does not converge to exactly zero. For this reason, to obtain more satisfactory results, the number of neurons in the hidden layer was increased to 50. The results are shown in the appendix B. In fact, the size of hidden neurons in Appendix B has been increased as follows to see the results:

- 30 hidden neurons
- 50 hidden neurons
- 100 hidden neurons
- 500 hidden neurons

Please refer to appendix B to see the results visualization. The running time for 500 neurons was very long. It took around 22 hours to train this neural network. At the end, it resulted in negligible enhancement in performance. The optimum number of hidden neurons is 50.

Using MATLAB shows that increasing the number of hidden neurons after 50 only slightly increases accuracy. It also reduces the speed of convergence. In the case of 500 hidden neurons, it took hours for the neural network to complete training.

Discussion

The neural network in both cases showed quite high accuracy. However, when comparing it with the results done by [1], some machine learning algorithms showed higher accuracy.

METHOD	UNIDIRECTIONAL	BIDIRECTIONAL
Linear Regression	98.23%	99.44%
K-NN	99.68%	99.9%
Decision Tree	99.96%	99.95%
Random Forest	99.95%	99.61%
Naïve Bayes	78%	97.55%
SVM RBF Kernel	97.96%	96.61%
SVM Linear Kernel	82.6%	98.5%

The artificial intelligence neural network used in this project achieved the following overall accuracy.

METHOD	UNIDIRECTIONAL	BIDIRECTIONAL
Neural Networks	99.78%	99.27%

Although the accuracy is high, as mentioned other machine learning techniques achieved higher accuracy. To achieve higher accuracy in neural networks, the following can be done with trial and error:

- Increase number of nodes
- Increase number of hidden layers
- Change parameters (activation function, Alpha, tolerance, … etc.)

Adjusting the network may increase or decrease accuracy. It may also change the speed at which such a network converges. Trail and error can take really a long time. As future improvement, trial and error may be avoided by constructing the neural network using heuristic methods, for example, use of neuro-evolution. Also, another future improvement is to test the neural network on other dataset to observe its performance on new attacks, as this is important given that a network expands and adopts new technologies.

References

[1] Hanan Hindy, Ethan Bayne, Miroslav Bures, Robert Atkinson3, Christos Tachtatzis, and Xavier Bellekens, “Machine Learning Based IoT Intrusion Detection System: An MQTT Case Study (MQTT-IoT-IDS2020 Dataset)”, Selected Papers from the 12th International Networking Conference, 2020.

[2] https://ieee-dataport.org/open-access/mqtt-iot-ids2020-mqtt-internet-things-intrusion-detection-dataset

[3] Hindy, H., Brosset, D., Bayne, E., Seeam, A.K., Tachtatzis, C., Atkinson, R.,Bellekens, X.: A taxonomy of network threats and the effect of current datasets on intrusion detection systems. IEEE Access, 2020.

[4] https://python-course.eu/machine-learning/dropout-neural-networks-in-python.php

[5]https://brilliant.org/wiki/backpropagation/#:~:text=Backpropagation%2C%20short%20for%20%22backward%20propagation,to%20the%20neural%20network's%20weights.

Appendix A

This appendix shows the python code of this project.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Aug 29 12:14:12 2019

@author: hananhindy
"""
import pandas as pd
import numpy as np
import os
import argparse
import pdb as debugger

from sklearn.preprocessing import OneHotEncoder
##from sklearn.linear_model import LogisticRegression
##from sklearn.neighbors import KNeighborsClassifier
##from sklearn.svm import SVC, LinearSVC
##from sklearn.naive_bayes import GaussianNB
##from sklearn.tree import DecisionTreeClassifier
##from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import classification_report

#import neural network
from sklearn.neural_network import MLPClassifier

# Helper Function
def str2bool(v):
    if v.lower() in ('yes', 'true', 't', 'y', '1'):
        return True
    elif v.lower() in ('no', 'false', 'f', 'n', '0'):
        return False
    else:
        raise argparse.ArgumentTypeError('Boolean value expected.')

#protocols = ['ARP', 'CDP', 'CLDAP', 'DATA', 'DNS', 'DTLS', 'DTP', 'ECHO', 'ICMP', 'ISAKMP','MDNS', 'NAT-
PMP', 'NBNS', 'NFS', 'NTP', 'PORTMAP', 'RADIUS', 'RIP', 'SRVLOC', 'SNMP',  'SSH', 'STP', 'TCP', 'UDP',
'XDMCP', 'MQTT', 'MPEG_PMT', 'MP2T', 'MPEG_PAT', 'DVB_SDT']
#label_encoder = LabelEncoder().fit(protocols)

one_hot_encoder = None

def load_file(path, mode, is_attack = 1, label = 1, folder_name='Ui/', sliceno = 0, verbose = True):
    #global label_encoder
    global one_hot_encoder
```

```
    #attacker_ips = ['192.168.2.5']

    columns_to_drop_packet = ['timestamp', 'src_ip', 'dst_ip', 'ip_flags', 'tcp_flags', 'mqtt_flags']
    columns_to_drop_uni = ['proto', 'ip_src', 'ip_dst']
    columns_to_drop_bi = ['proto', 'ip_src', 'ip_dst']

    if os.path.getsize(path)//10 ** 9 > 0:
        x = np.zeros((0,0))
        for chunk in pd.read_csv(path, chunksize=10 ** 6):
            chunk.drop(columns = columns_to_drop_packet, inplace = True)
            chunk = chunk[chunk.columns.drop(list(chunk.filter(regex='mqtt')))]

            chunk = chunk.fillna(-1)

            with open(folder_name + 'instances_count.csv','a') as f:
                f.write('{}, {} \n'.format(path, chunk.shape[0]))

            x_temp = chunk.loc[chunk['is_attack'] == is_attack]
            x_temp.drop('is_attack', axis = 1, inplace = True)
            #x_temp['protocol'] = label_encoder.transform(x_temp['protocol'])
            if one_hot_encoder == None:

                one_hot_encoder = OneHotEncoder(categorical_features=[0], n_values=30)
                x_temp = one_hot_encoder.fit_transform(x_temp).toarray()
            else:
                x_temp = one_hot_encoder.transform(x_temp).toarray()

            x_temp = np.unique(x_temp, axis = 0)

            if x.size == 0:
                x = x_temp
            else:
                x = np.concatenate((x, x_temp), axis = 0)
                x = np.unique(x, axis = 0)
    else:
        dataset = pd.read_csv(path)

        if mode == 1 or mode == 2:
            dataset = dataset.loc[dataset['is_attack'] == is_attack]
#           if is_attack == 0:
#                           dataset = dataset.loc[operator.and_(dataset['ip_src'].isin(attacker_ips) == False,
dataset['ip_dst'].isin(attacker_ips) == False)]
#           else:
#                                       dataset = dataset.loc[operator.or_(dataset['ip_src'].isin(attacker_ips),
dataset['ip_dst'].isin(attacker_ips))]
#
        if mode == 0:
```

```
        dataset.drop(columns=[columns_to_drop_packet], inplace = True)
        dataset = dataset[dataset.columns.drop(list(dataset.filter(regex='mqtt')))]
    elif mode == 1:
        dataset.drop(columns = columns_to_drop_uni, inplace = True)
    elif mode == 2:
        dataset.drop(columns = columns_to_drop_bi, inplace = True)

    if verbose:
        print(dataset.columns)

    dataset = dataset.fillna(-1)

    if mode == 0:
        x = dataset.loc[dataset['is_attack'] == is_attack]
        x.drop('is_attack', axis=1, inplace=True)
        #x['protocol'] = label_encoder.transform(x['protocol'])
        if one_hot_encoder == None:
            one_hot_encoder = OneHotEncoder(categorical_features=[0], n_values=30)
            x = one_hot_encoder.fit_transform(x).toarray()
        else:
            x = one_hot_encoder.transform(x).toarray()
    else:
        x = dataset.values

with open(folder_name + 'instances_count.csv','a') as f:
    f.write('all, {}, {} \n'.format(path, x.shape[0]))

x = np.unique(x, axis = 0)

with open(folder_name + 'instances_count.csv','a') as f:
    f.write('unique, {}, {} \n'.format(path, x.shape[0]))

if (mode == 1 and x.shape[0] > 100000) or (mode == 2 and x.shape[0] > 50000):
        temp = x.shape[0] // 10
        start = sliceno * temp
        end = start + temp - 1
        x = x[start:end,:]
        with open(folder_name + 'instances_count.csv','a') as f:
            f.write('Start, {}, End, {} \n'.format(start, end))
elif mode == 0:
    if x.shape[0] > 15000000:
        temp = x.shape[0] // 400
        start = sliceno * temp
        end = start + temp - 1
        x = x[start:end,:]
        with open(folder_name + 'instances_count.csv','a') as f:
            f.write('Start, {}, End, {} \n'.format(start, end))
    elif x.shape[0] > 10000000:
```

```
            temp = x.shape[0] // 200
            start = sliceno * temp
            end = start + temp - 1
            x = x[start:end,:]
            with open(folder_name + 'instances_count.csv','a') as f:
                f.write('Start, {}, End, {} \n'.format(start, end))
        elif x.shape[0] > 100000:
            temp = x.shape[0] // 10
            start = sliceno * temp
            end = start + temp - 1
            x = x[start:end,:]
            with open(folder_name + 'instances_count.csv','a') as f:
                f.write('Start, {}, End, {} \n'.format(start, end))


    y = np.full(x.shape[0], label)

    with open(folder_name + 'instances_count.csv','a') as f:
        f.write('slice, {}, {} \n'.format(path, x.shape[0]))

    return x, y

def classify_sub(classifier, x_train, y_train, x_test, y_test, cm_file_name, summary_file_name,
classifier_name, verbose = True):
    classifier.fit(x_train, y_train)
    pred = classifier.predict(x_test)

    cm = pd.crosstab(y_test, pred)
    cm.to_csv(cm_file_name)

    pd.DataFrame(classification_report(y_test, pred, output_dict = True,
zero_division=0)).transpose().to_csv(summary_file_name)

    if verbose:
        print(classifier_name + ' Done.\n')

    del classifier
    del pred
    del cm

def classify(random_state, x_train, y_train, x_test, y_test, folder_name, prefix = "", verbose = True):
    confusion_matrix_folder = os.path.join(folder_name, 'Confusion_Matrix/')
    summary_folder =  os.path.join(folder_name, 'Summary/')

    if os.path.isdir(confusion_matrix_folder) == False:
            os.mkdir(confusion_matrix_folder)
    if os.path.isdir(summary_folder) == False:
            os.mkdir(summary_folder)
```

```
  # 0- Nueral Net
  ann_classifier = MLPClassifier(activation='relu',solver='adam', alpha=10,
hidden_layer_sizes=(8,8,9,10, 2), random_state=random_state)
  # zip 1 > ann_classifier = MLPClassifier(activation='relu',solver='adam', alpha=0.1,
hidden_layer_sizes=(240,120), random_state=random_state)
  #ann_classifier = MLPClassifier(activation='tanh',solver='sgd', alpha=10, tol=0.00000001,
max_iter=3000, n_iter_no_change=5000, hidden_layer_sizes=(8,8,8),warm_start=True,
random_state=random_state,verbose=True)
  classify_sub(ann_classifier,
          x_train, y_train,
          x_test, y_test,
          confusion_matrix_folder + prefix + '_cm_ANN.csv',
          summary_folder + prefix + '_summary_ann.csv',
          'ANN',
          verbose)

##  # 1- Linear
##  linear_classifier = LogisticRegression(random_state = random_state)
##  classify_sub(linear_classifier,
##          x_train, y_train,
##          x_test, y_test,
##          confusion_matrix_folder + prefix + '_cm_linear.csv',
##          summary_folder + prefix + '_summary_linear.csv',
##          'Linear',
##          verbose)
##
##  # 2- KNN
##  knn_classifier = KNeighborsClassifier()
##  classify_sub(knn_classifier,
##          x_train, y_train,
##          x_test, y_test,
##          confusion_matrix_folder + prefix + '_cm_knn.csv',
##          summary_folder + prefix + '_summary_knn.csv',
##          'KNN',
##          verbose)
##
##  #3- RBF SVM
##  kernel_svm_classifier = SVC(kernel = 'rbf', random_state = random_state, gamma='scale')
##  classify_sub(kernel_svm_classifier,
##          x_train, y_train,
##          x_test, y_test,
##          confusion_matrix_folder + prefix + '_cm_kernel_svm.csv',
##          summary_folder + prefix + '_summary_kernel_svm.csv',
##          'SVM',
##          verbose)
##
##  #4- Naive Bayes
```

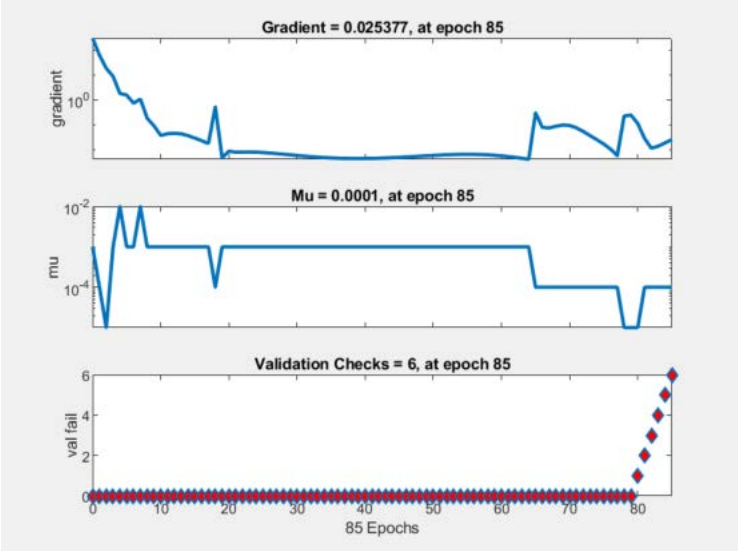
```
##    naive_classifier = GaussianNB()
##    classify_sub(naive_classifier,
##                 x_train, y_train,
##                 x_test, y_test,
##                 confusion_matrix_folder + prefix + '_cm_naive.csv',
##                 summary_folder + prefix + '_summary_naive.csv',
##                 'Naive',
##                 verbose)
##
##    #5- Decision Tree
##          decision_tree_classifier = DecisionTreeClassifier(criterion = 'entropy', random_state =
random_state)
##    classify_sub(decision_tree_classifier,
##                 x_train, y_train,
##                 x_test, y_test,
##                 confusion_matrix_folder + prefix + '_cm_decision_tree.csv',
##                 summary_folder + prefix + '_summary_decision_tree.csv',
##                 'Decision Tree',
##                 verbose)
##
##    #6- Random Forest
##       random_forest_classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy',
random_state = random_state)
##    classify_sub(random_forest_classifier,
##                 x_train, y_train,
##                 x_test, y_test,
##                 confusion_matrix_folder + prefix + '_cm_random_forest.csv',
##                 summary_folder + prefix + '_summary_random_forest.csv',
##                 'Random Forest',
##                 verbose)
##
##    # 7- Linear SVM
##    svm_classifier = LinearSVC(random_state = random_state)
##    classify_sub(svm_classifier,
##                 x_train, y_train,
##                 x_test, y_test,
##                 confusion_matrix_folder + prefix + '_cm_svm.csv',
##                 summary_folder + prefix + '_summary_svm.csv',
##                 'SVM',
##                 verbose)

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--mode', type = int, default = 1)
    parser.add_argument('--output', default='Classification_Bi')
    parser.add_argument('--verbose', type = str2bool, default = True)

    args = parser.parse_args()
```

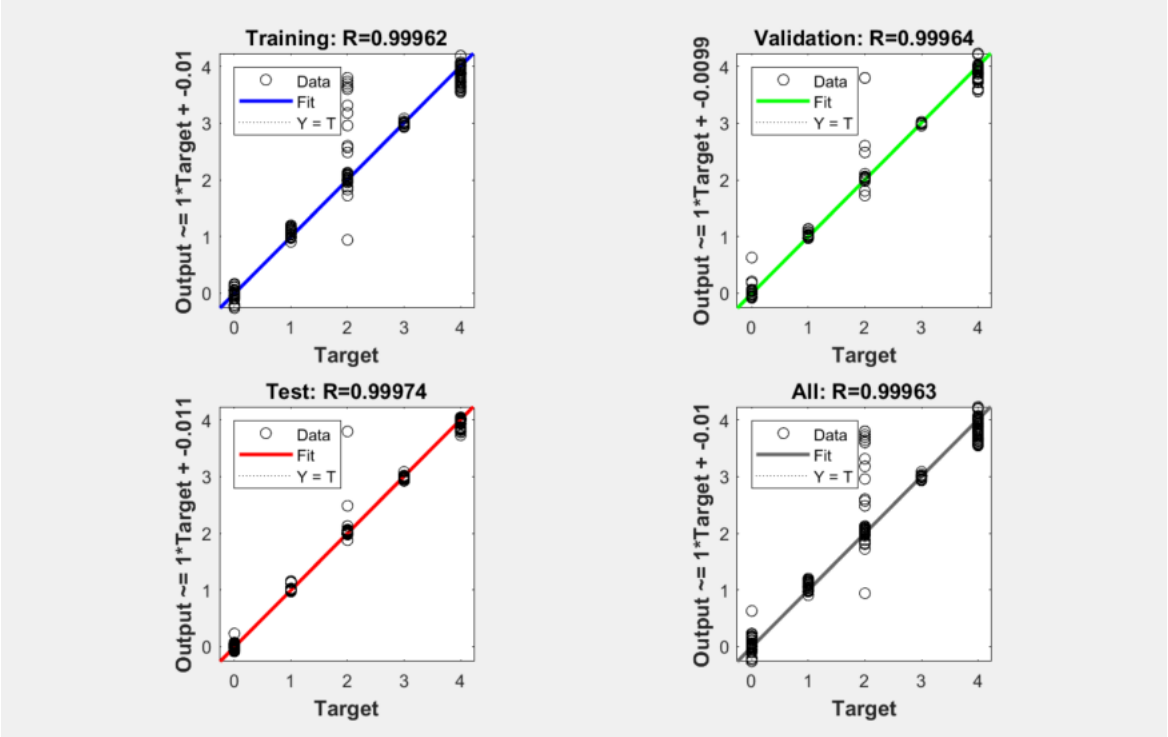
```
for slice_number in range(10):
    prefix = ''
    if args.mode == 1:
        prefix = 'uniflow_'
    elif args.mode == 2:
        prefix = 'biflow_'

    if args.verbose:
        print('Starting Slice #: {}'.format(slice_number))
        print('Start Classification')

    random_state = 0
    folder_name = '{}_{}/'.format(args.output, slice_number)

    if os.path.isdir(folder_name) == False:
        os.mkdir(folder_name)

    x, y = load_file(prefix + 'normal.csv',
                     args.mode,
                     0, 0,
                     folder_name,
                     slice_number,
                     args.verbose)

    x_temp, y_temp = load_file(prefix + 'scan_A.csv',
                               args.mode,
                               1, 1,
                               folder_name,
                               slice_number,
                               args.verbose)

    x = np.concatenate((x, x_temp), axis = 0)
    y = np.append(y, y_temp)
    del x_temp, y_temp

    x_temp, y_temp = load_file(prefix + 'scan_sU.csv',
                               args.mode,
                               1, 2,
                               folder_name,
                               slice_number,
                               args.verbose)

    x = np.concatenate((x, x_temp), axis = 0)
    y = np.append(y, y_temp)
    del x_temp, y_temp
```

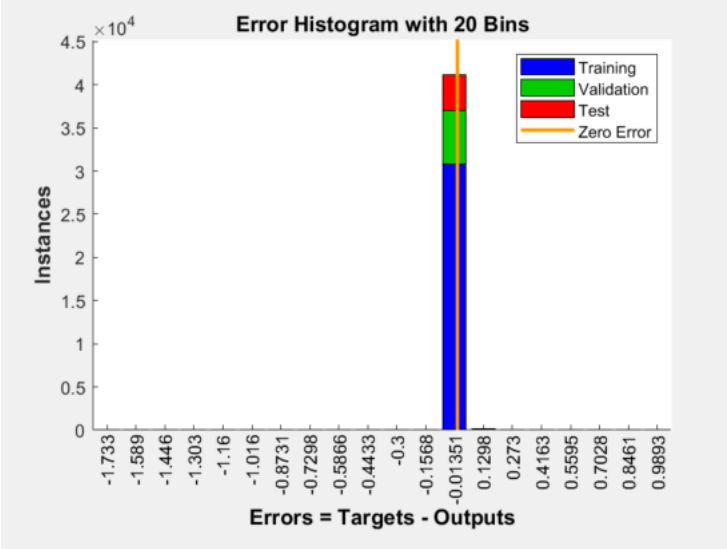
```
x_temp, y_temp = load_file(prefix + 'sparta.csv',
                    args.mode,
                    1, 3,
                    folder_name,
                    slice_number,
                    args.verbose)

x = np.concatenate((x, x_temp), axis = 0)
y = np.append(y, y_temp)
del x_temp, y_temp

x_temp, y_temp = load_file(prefix + 'mqtt_bruteforce.csv',
                    args.mode,
                    1, 4,
                    folder_name,
                    slice_number,
                    args.verbose)

x = np.concatenate((x, x_temp), axis = 0)
y = np.append(y, y_temp)
del x_temp, y_temp



x_train, x_test, y_train, y_test = train_test_split(x, y,
                                    test_size = 0.25,
                                    random_state = 42)

#Up till here, the x and y sets are created and ready for use

classify(random_state, x_train, y_train, x_test, y_test,
      folder_name, "slice_{}_no_cross_validation".format(slice_number), args.verbose)

kfold = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 0)

counter = 0
for train, test in kfold.split(x, y):
   classify(random_state, x[train], y[train], x[test], y[test],
         folder_name, "slice_{}_k_{}".format(slice_number, counter), args.verbose)
   counter += 1

del x
del y
del x_train
del x_test
del y_train
del y_test
```

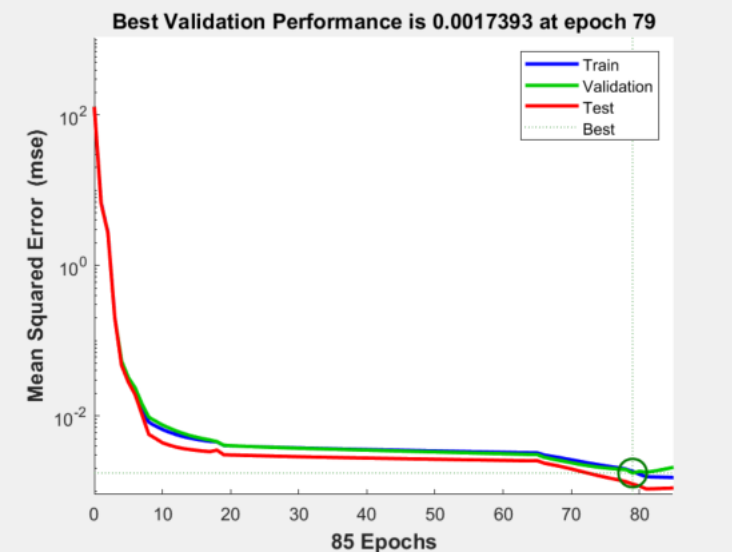
Appendix B

Results of 30 neurons







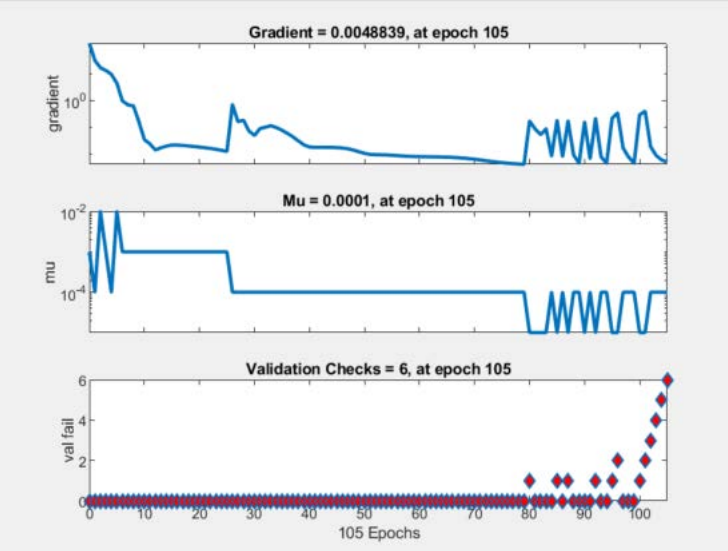


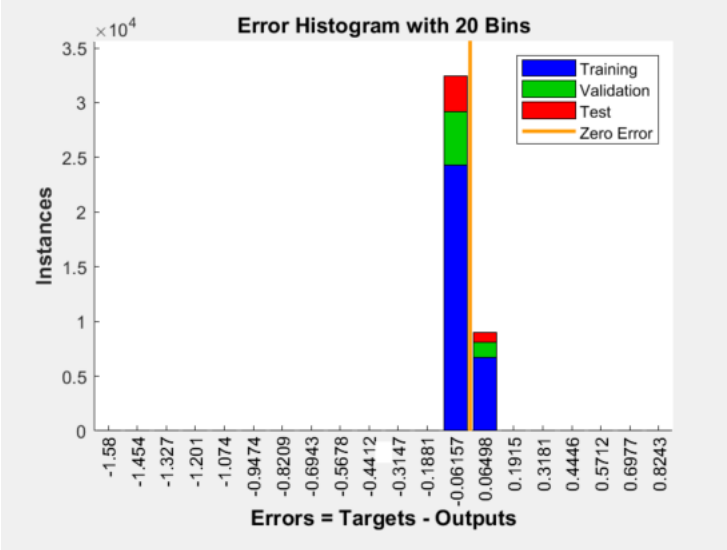
Results

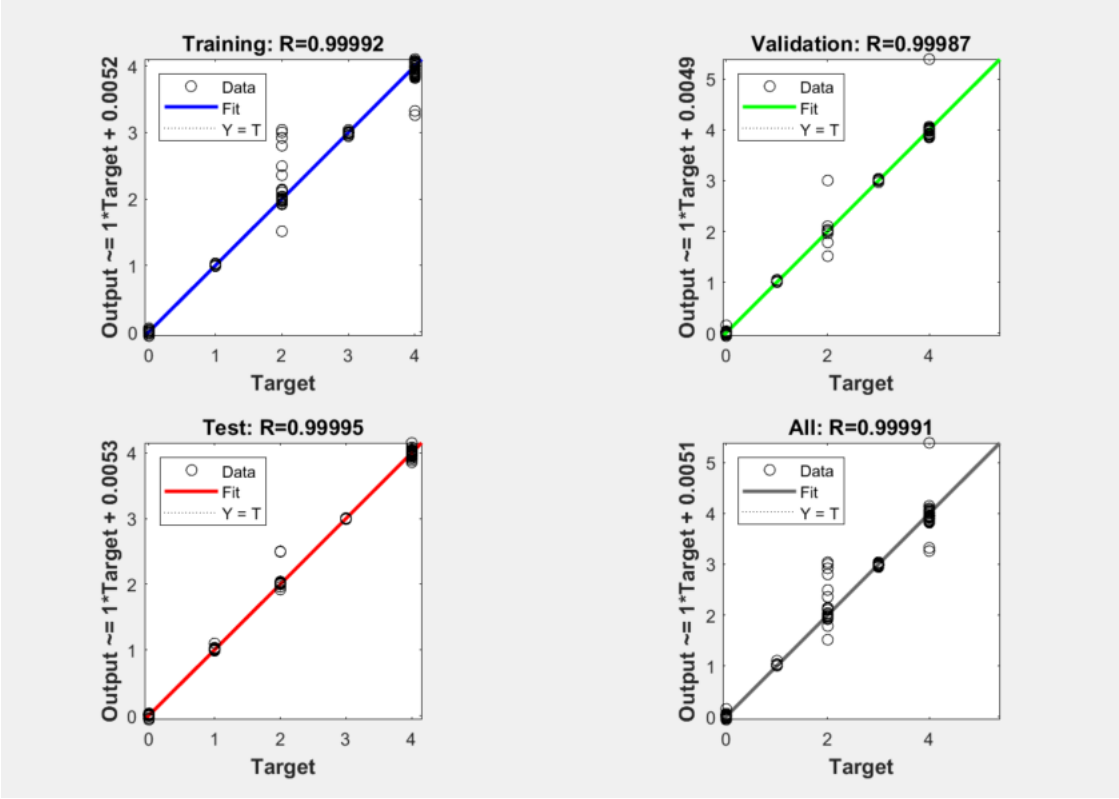
	Samples	MSE	R
Training:	31119	1.83481e-3	9.99618e-1
Validation:	6224	1.73926e-3	9.99640e-1
Testing:	4149	1.24236e-3	9.99740e-1

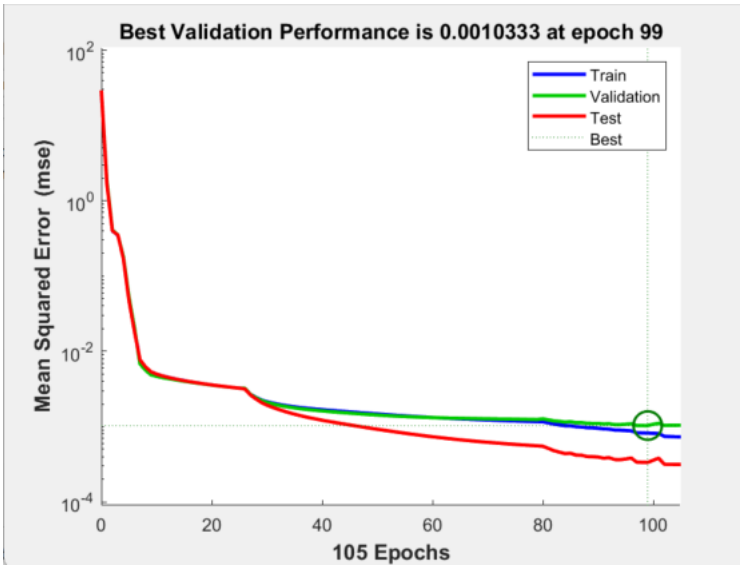
Plot Fit | Plot Error Histogram | Plot Regression

Results of 50 neurons

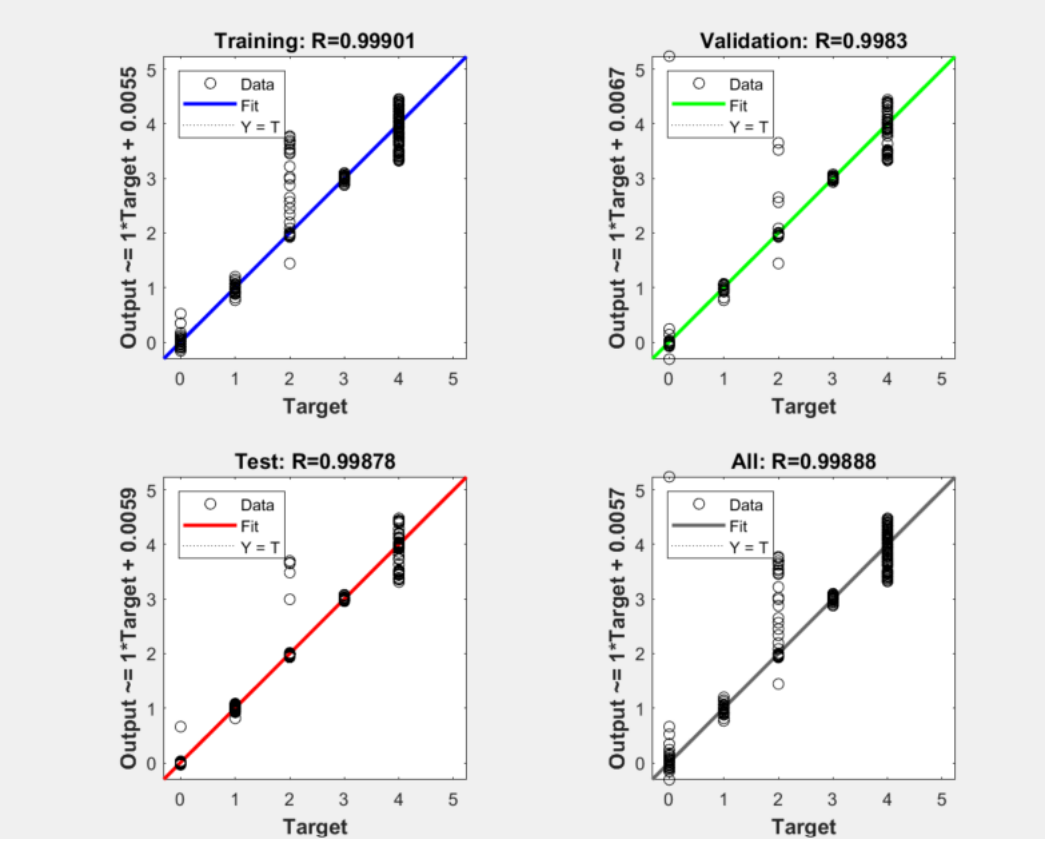


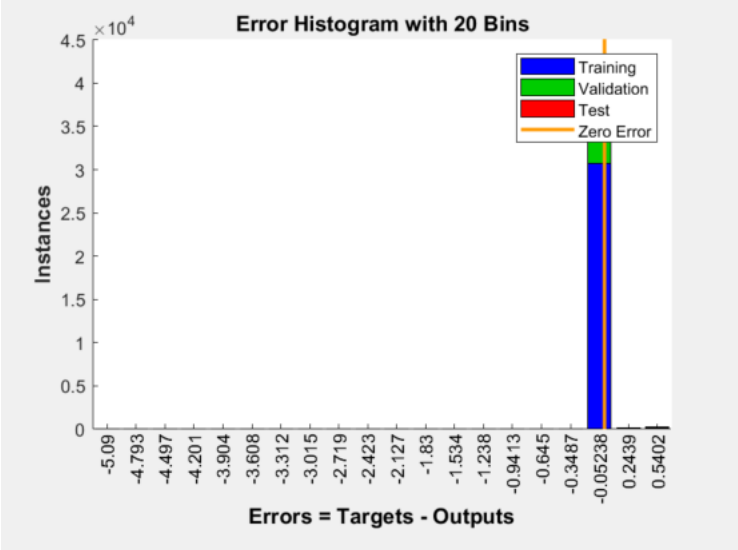


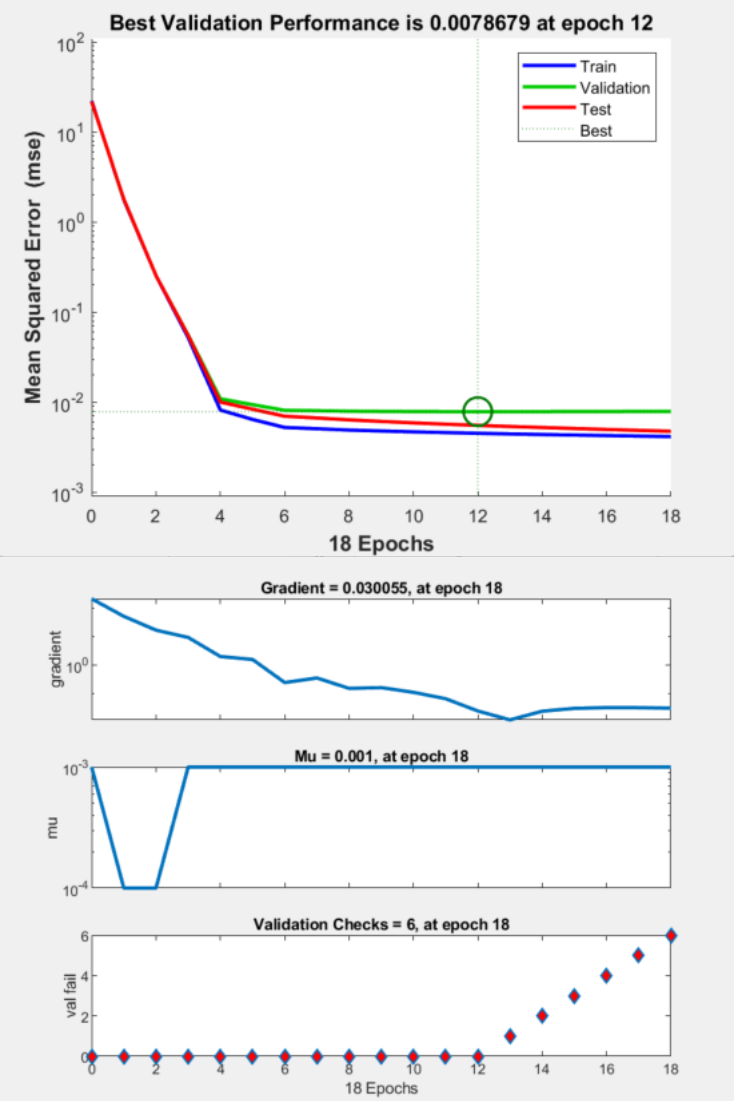

Training: R=0.99992
Output ~= 1*Target + 0.0052
Target
Data
Fit
Y = T
Validation: R=0.99987
Output ~= 1*Target + 0.0049
Target
Data
Fit
Y = T
Test: R=0.99995
Output ~= 1*Target + 0.0053
Target
Data
Fit
Y = T
All: R=0.99991
Output ~= 1*Target + 0.0051
Target
Data
Fit
Y = T


Best Validation Performance is 0.0010333 at epoch 99
Mean Squared Error (mse)
105 Epochs
Train
Validation
Test
Best

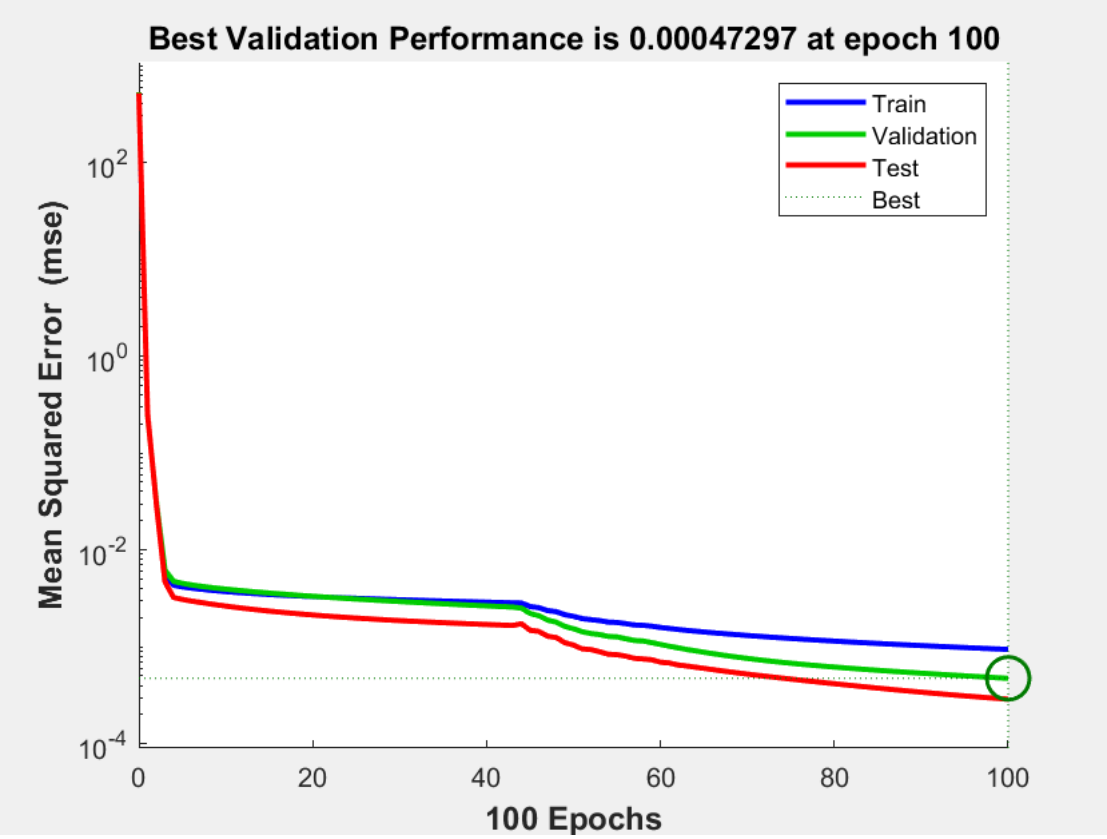
Results for 100 hidden neurons size



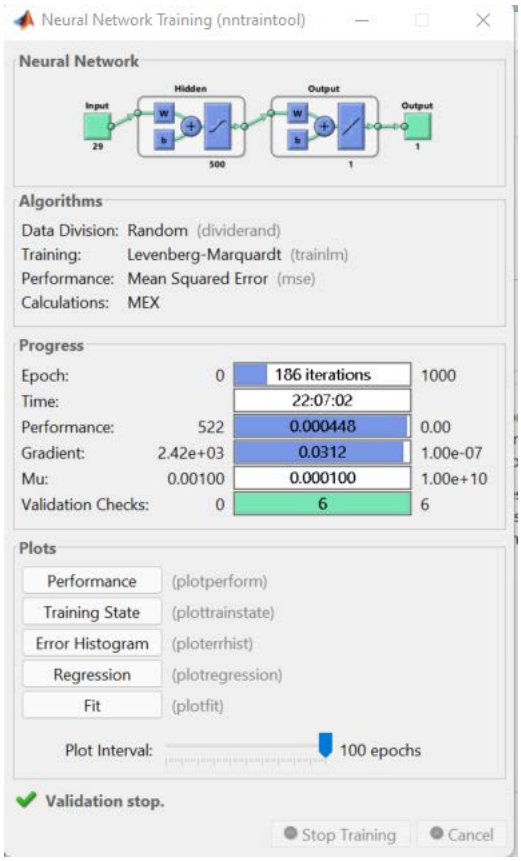


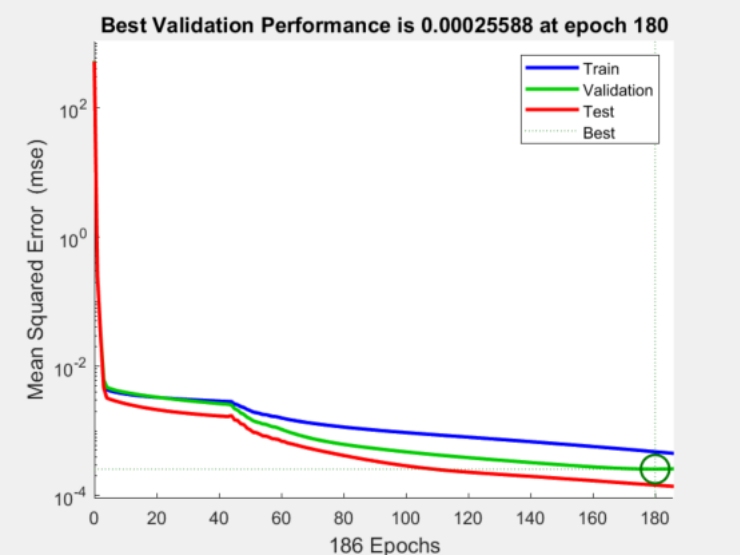

Best Validation Performance is 0.0078679 at epoch 12
Mean Squared Error (mse)
Train
Validation
Test
Best
18 Epochs
Gradient = 0.030055, at epoch 18
gradient
Mu = 0.001, at epoch 18
mu
Validation Checks = 6, at epoch 18
val fail
18 Epochs

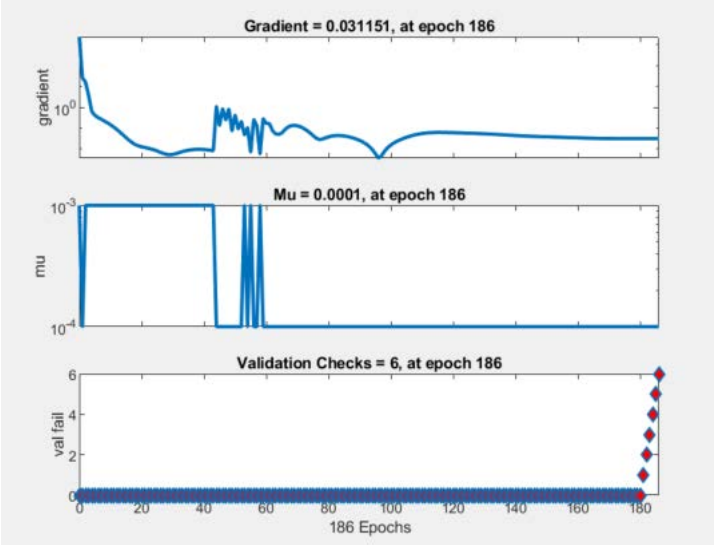
Results for 500 hidden neurons

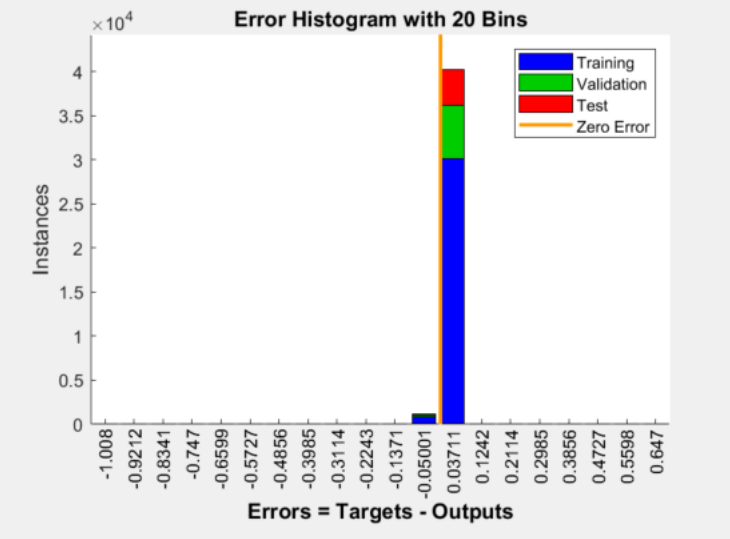


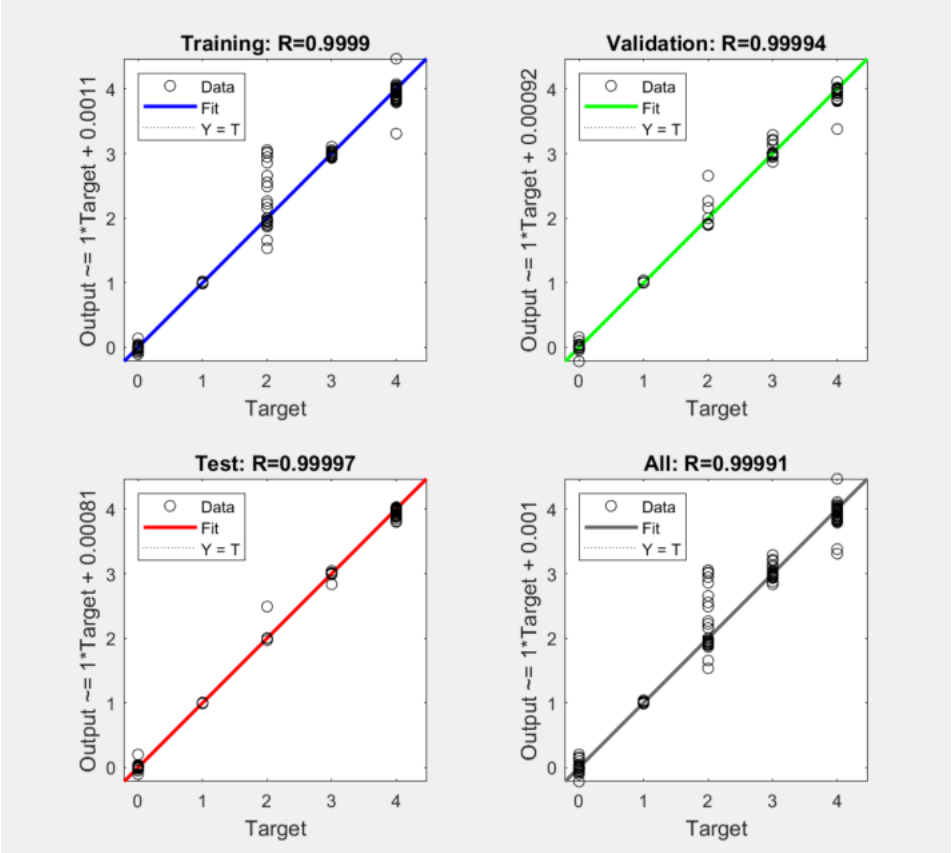
after only 100 iterations of epoch


Neural Network Training (nntraintool)
Neural Network
Input
29
Hidden
W
b
500
Output
W
b
1
Output
1
Algorithms
Data Division: Random (dividerand)
Training: Levenberg-Marquardt (trainlm)
Performance: Mean Squared Error (mse)
Calculations: MEX
Progress
Epoch: 0 186 iterations 1000
Time: 22:07:02
Performance: 522 0.000448 0.00
Gradient: 2.42e+03 0.0312 1.00e-07
Mu: 0.00100 0.000100 1.00e+10
Validation Checks: 0 6 6
Plots
Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Regression (plotregression)
Fit (plotfit)
Plot Interval: 100 epochs
Validation stop.
Stop Training
Cancel


Best Validation Performance is 0.00025588 at epoch 180
Train
Validation
Test
Best
Mean Squared Error (mse)
10^2
10^0
10^-2
10^-4
0
20
40
60
80
100
120
140
160
180
186 Epochs


Gradient = 0.031151, at epoch 186
gradient
Mu = 0.0001, at epoch 186
mu
Validation Checks = 6, at epoch 186
val fail
186 Epochs


Error Histogram with 20 Bins
Training
Validation
Test
Zero Error
Instances
Errors = Targets - Outputs


Training: R=0.9999
Data
Fit
Y = T
Output ~= 1*Target + 0.0011
Target
Validation: R=0.99994
Data
Fit
Y = T
Output ~= 1*Target + 0.00092
Target
Test: R=0.99997
Data
Fit
Y = T
Output ~= 1*Target + 0.00081
Target
All: R=0.99991
Data
Fit
Y = T
Output ~= 1*Target + 0.001
Target