APPENDIX

CONVERT TO FUZZY EDGE IMAGE

```
import os,sys
from PIL import Image as Im
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import cv2
import statistics
for num in range(1,57):
    img = mpimg.imread('dataset-vin/vin%d.jpg' % (num),
0)

    width, height = img.shape[:2]
    FEimg = np.zeros_like(img)
    grayimg = np.zeros_like(img)
    delta1=64
    for x in range(0,width):
        for y in range(0,height):
            sample = img[x][y]
            gray = np.amax(sample)
            grayimg[x][y] = [gray,gray,gray]
    for x in range(1, width-1):
        for y in range(1, height-1):
            arr_g = np.array([])
```

```
            g = grayimg[x][y][0]
            for i in range(x-1, x+2):
                for j in range(y-1, y+2):
                    sample = grayimg[i][j][0]
                    arr_g = np.append(arr_g, sample)
            gmax = np.amax(arr_g)
            gmin = np.amin(arr_g)
            val1 = float((gmax-gmin)/delta1 )
            m1 = np.array([1,val1])
            res1 = np.amin(m1)
            gmean = np.mean(arr_g)
            val2 = (g-gmean)/delta1
            m2 = np.array([1,val2])
            res2 = 1- np.amin(m2)
            arr_f = np.array([res1,res2])
            fmin = np.amin(arr_f)
            fmin = fmin*255
            if(g == 0):
                result=0
            else:
                result = (fmin/g)
            if(result >=1):
                result=1
            fuz = int(result*255)
            FEimg[x][y] = [fuz,fuz,fuz]
        ImFE = Im.fromarray(FEimg)
```

```
ImFE.save('chassis_fuzzy/chassis_fedge/fe_vin%d.jpg' % (num),
0)
    print("done")
```

NEURAL NETWORK CALL FUNCTION

```
    import os,sys
    from PIL import Image as Im
    import math
    import numpy as np
    import matplotlib.pyplot as plt
    import matplotlib.image as mpimg
    import cv2
    import statistics
    import time
    import networkabc
    import networknum
    import gzip

    # Original source
http://neuralnetworksanddeeplearning.com/chap1.html
    # Credit: Michael Nielsen, Jun 2019

    def getred(height, width, img):
        red = np.array(([]))
        for i in range(0,height):
            for j in range(0,width):
```

```
                sample = img[i][j]
                if(sample[0] > 0):
                    val = 1
                else:
                    val = 0
                red = np.append(red, val)
        return red

    def load_data_wrapper(tr_d, te_d, nodes):
        training_inputs = [np.reshape(x, (594, 1)) for x in
tr_d[0]]
        training_results = [vectorized_result(y, nodes) for y
in tr_d[1]]
        training_data = zip(training_inputs,
training_results)
        test_inputs = [np.reshape(x, (594, 1)) for x in
te_d[0]]
        test_data = zip(test_inputs, te_d[1])
        return (training_data, test_data)

    def vectorized_result(j, nodes):
        """Return a 10-dimensional unit vector with a 1.0 in
the jth
        position and zeroes elsewhere.  This is used to
convert a digit
        (0...9) into a corresponding desired output from the
neural
        network."""
```

```
        e = np.zeros((nodes, 1))
        e[int(j)] = int(1.0)
        return e

    a =
np.loadtxt("FedgeSegment/neuralnet_data/train_num.txt")
    b =
np.loadtxt("FedgeSegment/neuralnet_data/target_num.txt")

    train_tup_num = (a,b)

    c =
np.loadtxt("FedgeSegment/neuralnet_data/test_num.txt")
    d =
np.loadtxt("FedgeSegment/neuralnet_data/test_target_num.txt")

    test_tup_num = (c,d)

    e =
np.loadtxt("FedgeSegment/neuralnet_data/train_abc.txt")
    f =
np.loadtxt("FedgeSegment/neuralnet_data/target_abc.txt")

    train_tup_abc = (e,f)

    g =
np.loadtxt("FedgeSegment/neuralnet_data/test_abc.txt")
```

```
    h =
np.loadtxt("FedgeSegment/neuralnet_data/test_target_abc.txt")

    test_tup_abc = (g,h)

    training_data, test_data =
load_data_wrapper(train_tup_num, test_tup_num, 10)
    training_data2, test_data2 =
load_data_wrapper(train_tup_abc, test_tup_abc, 26)

    NN = networknum.Network([594,30,10])
    NN2 = networkabc.Network([594,30,26])

    NN.SGD(training_data, 4000, 10, 3.0, test_data=test_data)

    print('')
    print('')

    NN2.SGD(training_data2, 4000, 15, 3.0,
test_data=test_data2)

    DIRpr = "FedgeSegment/TestSample/vin/"
    for w in os.listdir(DIRpr):
        resultp = ''
        DIRpr2 = DIRpr + str(w)
        namesave = str(w) + ".txt"
        for x in os.listdir(DIRpr2):
```

```
            DIRpr3 = DIRpr2 + "/" + str(x)
            for y in os.listdir(DIRpr3):
                DIRpr4 = DIRpr3 + "/" + str(y)
                test_img = mpimg.imread(DIRpr4)
                h,w = test_img.shape[:2]
                red = getred(h,w,test_img)
                inp = [np.reshape(red, (594,1))]
                inp_data = zip(inp, [0])
                if('letter' in x):
                    r = NN2.evaluate(inp_data,1)
                    alph = 'abcdefghijklmnopqrstuvwxyz'
                    print("r = " + str(r))
                    r_txt = alph[r[0]]
                elif('number' in x):
                    r = NN.evaluate(inp_data,1)
                    numeric = '0123456789'
                    print("r = " + str(r))
                    r_txt = numeric[r[0]]
                resultp = resultp + r_txt
        txtsavedir = "FedgeSegment/TestSample/test_results/"
+ namesave
        np.savetxt(txtsavedir, [resultp], fmt="%s")
```

NEURAL NETWORK USED FOR RECOGNITION

CREATED BY Nielsen[6]

```
"""
network.py
~~~~~~~~~~
A module to implement the stochastic gradient descent learning
algorithm for a feedforward neural network.  Gradients are calculated
using backpropagation.  Note that I have focused on making the code
simple, easily readable, and easily modifiable.  It is not optimized,
and omits many desirable features.
"""

#### Libraries
# Standard library
import random

# Third-party libraries
import numpy as np

class Network(object):

    def __init__(self, sizes):
```

```
        """The list ``sizes`` contains the number of
neurons in the
        respective layers of the network.  For example,
if the list
        was [2, 3, 1] then it would be a three-layer
network, with the
        first layer containing 2 neurons, the second
layer 3 neurons,
        and the third layer 1 neuron.  The biases and
weights for the
        network are initialized randomly, using a
Gaussian
        distribution with mean 0, and variance 1.  Note
that the first
        layer is assumed to be an input layer, and by
convention we
        won't set any biases for those neurons, since
biases are only
        ever used in computing the outputs from later
layers."""
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in
sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1],
sizes[1:])]
        self.currentW = self.weights
        self.currentB = self.biases
```

```
    def feedforward(self, a):
        """Return the output of the network if ``a`` is
input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a


    def SGD(self, training_data, epochs, mini_batch_size,
eta,
            test_data=None):
        """Train the neural network using mini-batch
stochastic
        gradient descent.  The ``training_data`` is a
list of tuples
        ``(x, y)`` representing the training inputs and
the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If ``test_data`` is provided
then the
        network will be evaluated against the test data
after each
        epoch, and partial progress printed out.  This is
useful for
        tracking progress, but slows things down
substantially."""
        if test_data: n_test = len(test_data)
```

```
n = len(training_data)
for j in xrange(epochs):
    random.shuffle(training_data)
    mini_batches = [
        training_data[k:k+mini_batch_size]
        for k in xrange(0, n, mini_batch_size)]
    for mini_batch in mini_batches:
        self.update_mini_batch(mini_batch, eta)
    if test_data:
        k1 = self.evaluate(test_data)
        print "Epoch {0}: {1} / {2}".format(
            j, k1, n_test)
        if(j==0):
            self.bestW = self.weights
            self.bestB = self.biases
            k2 = k1

        elif(j>0 and k1 > k2):
            self.bestW = self.weights
            self.bestB = self.biases
            k2=k1

        elif(j == (epochs-1) and k1<k2):
            self.weights = self.bestW
            self.biases = self.bestB
```

```
                        print "rollback to the best state
(  {0} / {1}  )".format(k2, n_test)

                else:
                    print "Epoch {0} complete".format(j)

        def update_mini_batch(self, mini_batch, eta):
            """Update the network's weights and biases by
applying
            gradient descent using backpropagation to a
single mini batch.
            The ``mini_batch`` is a list of tuples ``(x,
y)``, and ``eta``
            is the learning rate."""
            nabla_b = [np.zeros(b.shape) for b in
self.biases]
            nabla_w = [np.zeros(w.shape) for w in
self.weights]
            for x, y in mini_batch:
                delta_nabla_b, delta_nabla_w =
self.backprop(x, y)
                nabla_b = [nb+dnb for nb, dnb in zip(nabla_b,
delta_nabla_b)]
                nabla_w = [nw+dnw for nw, dnw in zip(nabla_w,
delta_nabla_w)]
            self.weights = [w-(eta/len(mini_batch))*nw
                            for w, nw in zip(self.weights,
nabla_w)]
```

```
            self.biases = [b-(eta/len(mini_batch))*nb
                           for b, nb in zip(self.biases,
nabla_b)]


        def backprop(self, x, y):
            """Return a tuple ``(nabla_b, nabla_w)``
representing the
            gradient for the cost function C_x.  ``nabla_b``
and
            ``nabla_w`` are layer-by-layer lists of numpy
arrays, similar
            to ``self.biases`` and ``self.weights``."""
            nabla_b = [np.zeros(b.shape) for b in
self.biases]
            nabla_w = [np.zeros(w.shape) for w in
self.weights]
            activation = x
            activations = [x]
            zs = []
            for b, w in zip(self.biases, self.weights):
                z = np.dot(w, activation)+b
                zs.append(z)
                activation = sigmoid(z)
                activations.append(activation)
            delta = self.cost_derivative(activations[-1], y)
* \
                sigmoid_prime(zs[-1])
            nabla_b[-1] = delta
```

```
            nabla_w[-1] = np.dot(delta, activations[-
2].transpose())

            for l in xrange(2, self.num_layers):
                z = zs[-l]
                sp = sigmoid_prime(z)
                delta = np.dot(self.weights[-
l+1].transpose(), delta) * sp
                nabla_b[-l] = delta
                nabla_w[-l] = np.dot(delta, activations[-l-
1].transpose())
            return (nabla_b, nabla_w)

        def evaluate(self, test_data, p=None):
            """Return the number of test inputs for which the
neural
            network outputs the correct result. Note that the
neural
            network's output is assumed to be the index of
whichever
            neuron in the final layer has the highest
activation."""
            if(p == None):
                test_results =
[(np.argmax(self.feedforward(x)), y)
                                for (x, y) in test_data]
                return sum(int(x == y) for (x, y) in
test_results)
```

```
            else:
                test_results =
[(np.argmax(self.feedforward(x)),y) for (x,y) in test_data]
                print(len(test_data))
                print(test_results)
                return test_results[0]

        def predict(self,test_data):
            result = self.feedforward(test_data)
            m = np.argmax(result)
            print(m)
            ind = np.unravel_index(np.argmax(result,
axis=None), result.shape)
            print(ind)
            return m

        def cost_derivative(self, output_activations, y):
            """Return the vector of partial derivatives
\partial C_x /
            \partial a for the output activations."""
            return (output_activations-y)
```

CODE FOR THRESHOLDING OF FUZZY EDGE IMAGES

```
import os,sys

import cv2

import numpy as np

from matplotlib import pyplot as plt

import matplotlib.image as mpimg


DIR = "chassis_fuzzy/chassis_fedge/"

for source in os.listdir(DIR):

    DIR2 = DIR + str(source)

    img = cv2.imread(DIR2, 0)

    maxi = np.amax(img)

    av = np.average(img)

    thres = 5 + ((maxi+av)/2)

    ret,thresh1 =
cv2.threshold(img,thres,255,cv2.THRESH_BINARY)

    savedir = "fedge_binary/vin/" + str(source)

    cv2.imwrite(savedir, thresh1)


print("done")
```

CODE FOR COMPRESSING THE DATASET AND DATASET LABELS

```
import os,sys
from PIL import Image as Im
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import cv2
import statistics
import time

def getred(height, width, img):
    red = np.array(([]))
    for i in range(0,height):
        for j in range(0,width):
            sample = img[i][j]
            if(sample[0] > 0):
                val = 1
            else:
                val = 0
            red = np.append(red, val)
    return red
```

```
def getdata(x):
    data = np.array(([]), dtype=float)
    target = np.array(([]), dtype=int)
    for i in os.listdir(x):
        DIR2 = x + str(i) + "/"
        if len([j for j in os.listdir(DIR2)]) > 0:
            for j in os.listdir(DIR2):
                imgdir = DIR2 + str(j)
                img = mpimg.imread(imgdir)
                h,w = img.shape[:2]
                flat = getred(h,w,img)
                target = np.append(target, int(i))
                if(len(data) == 0):
                    data = [flat]
                else:
                    red = [flat]
                    data = np.vstack((data, red))
    return data,target


DIR = "FedgeSegment/TrainSample/samples_abc/"
trdata_abc, trlabel_abc = getdata(DIR)
```

```
print('saving training letter data')
np.savetxt("FedgeSegment/neuralnet_data/train_abc.txt",
trdata_abc, fmt="%s")
np.savetxt("FedgeSegment/neuralnet_data/target_abc.txt"
, trlabel_abc, fmt="%s")
print("done")

DIR = "FedgeSegment/TrainSample/samples_num/"
trdata_num, trlabel_num = getdata(DIR)
print('saving training number data')
np.savetxt("FedgeSegment/neuralnet_data/train_num.txt",
trdata_num, fmt="%s")
np.savetxt("FedgeSegment/neuralnet_data/target_num.txt"
, trlabel_num, fmt="%s")
print("done")

DIR = "FedgeSegment/TestSample/samples_abc/"
tedata_abc, telabel_abc = getdata(DIR)
print('saving testing letter data')
np.savetxt("FedgeSegment/neuralnet_data/test_abc.txt",
tedata_abc, fmt="%s")
np.savetxt("FedgeSegment/neuralnet_data/test_target_abc
.txt", telabel_abc, fmt="%s")
print("done")
```

```
DIR = "FedgeSegment/TestSample/samples_num/"
tedata_num, telabel_num = getdata(DIR)
print('saving testing number data')
np.savetxt("FedgeSegment/neuralnet_data/test_num.txt",
tedata_num, fmt="%s")
np.savetxt("FedgeSegment/neuralnet_data/test_target_num
.txt", telabel_num, fmt="%s")
print("done")
```

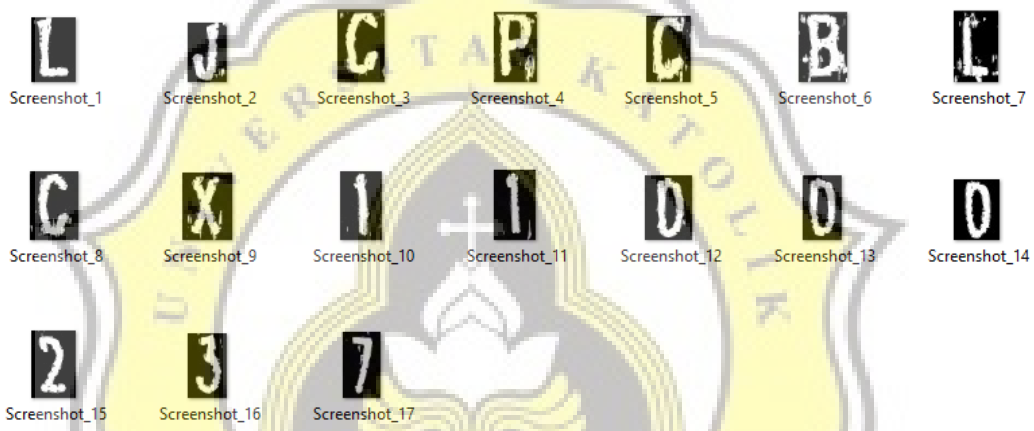
EDGE DETECTION KERNEL CODE

```
import os,sys
from PIL import Image as Im
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import cv2
import statistics
from decimal import *


DIR = ('chassis_fuzzy/chassis_fedge/')
fulldir = "E:/daftar ide skripsi/Project-PraProject/" + DIR
for source in os.listdir(fulldir):
    DIR2 = DIR + str(source)
    img = mpimg.imread(DIR2,0)

    width, height = img.shape[:2]

    gray_img = np.zeros_like(img)

    result_img = np.zeros_like(img)
```

```
    kernel =
[[0.13,0.13,0.13],[0.13,0.13,0.13],[0.13,0.13,0.13]]



    for x in range(0, width):
        for y in range(0, height):
            colorVal=0
            for z in range(0,2):
                colorVal += img[x][y][z]
                avg = colorVal/3
                gray_img[x][y] = [avg,avg,avg]


    for x in range(1, width-1):
        for y in range(1, height-1):
            edge = (kernel[0][0] * gray_img[x-1][y-1][0]) + \
                (kernel[1][0] * gray_img[x][y-1][0]) + \
                (kernel[2][0] * gray_img[x+1][y-1][0]) + \
                (kernel[0][1] * gray_img[x-1][y][0]) + \
                (kernel[1][1] * gray_img[x][y][0]) + \
                (kernel[2][1] * gray_img[x+1][y][0]) + \
                (kernel[0][2] * gray_img[x-1][y+1][0]) + \
                (kernel[1][2] * gray_img[x][y+1][0]) + \
                (kernel[2][2] * gray_img[x+1][y+1][z])

            if(edge>=128):
```

```
            edge = 255
        else:
            edge=0
        result_img[x][y] = [edge,edge,edge]

ImE = Im.fromarray(result_img)
ImE.save('fuzzyfiltered/' + str(source), 0)
```

Images



The results of the image processing steps can be seen above. From the original image, fuzy edge map, binarized map, and segmented characters.

UNICHECK

Submission author:
16k10027 I WAYAN ARI WIJAYA

Check ID:
15991960

Check date:
17.01.2020 06:44:58 GMT+0

Check type:
Doc vs Internet + Library

Report date:
17.01.2020 07:01:26 GMT+0

User ID:
29151

Unika SOEGIJAPRANATA
Talenta pro patria et humanitate

File name: **16.K1.0027_I Wayan Ari Wijaya.docx**

File ID: **20292572** Page count: **13** Word count: **4681** Character count: **27336** File size: **66.62 KB**

0.94% Matches

Highest match: **0.21%** with source **http://docshare.tips/thermal-imaging-cameras_592c437bee3435d477991d36.html**

0.94% Internet Matches 93 ..... Page 15

No Library Sources Found

0% Quotes

No quotes found

0% Exclusions

No exclusions found

Replacement

Character replacement 3