Assignment 3

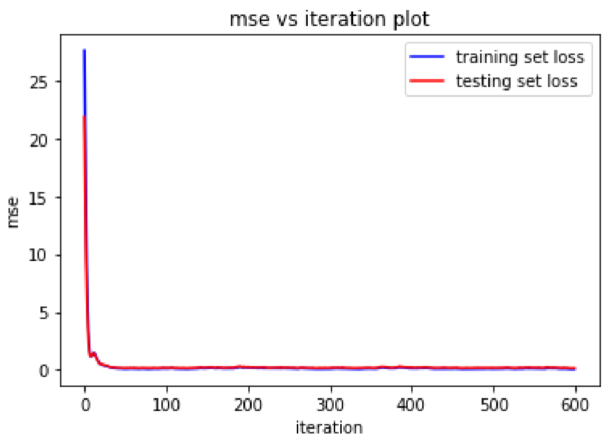
1. Convolutional Neural Network

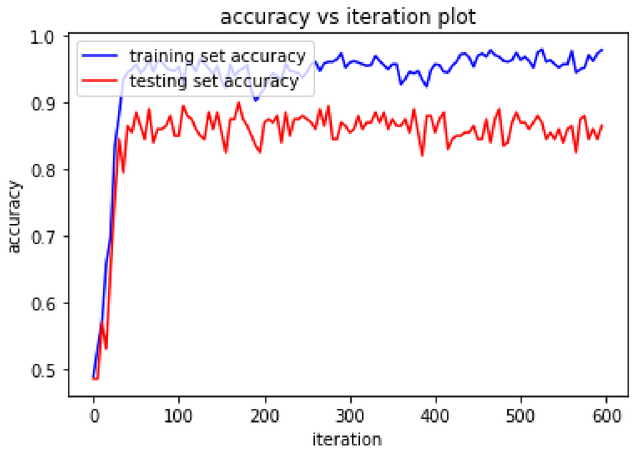
```
from scipy.io import loadmat
from keras.models import Sequential
from keras.layers import Dense, Activation, Conv1D, Flatten, AveragePooling1D
import numpy as np
from preprocessing import NormalScaler
import matplotlib.pyplot as plt
import keras

# loading data
data = loadmat('./data_for_cnn.mat')['ecg_in_window'].astype(np.float64)
data_labels = loadmat('./class_label.mat')['label'].astype(np.int)

data = np.concatenate((data, data_labels), axis=1)

np.random.shuffle(data)

# data preprocessing
scaler = NormalScaler()
for j in range(data.shape[1]-1):
    scaler.fit(data[:,j])
    data[:,j] = scaler.transform(data[:,j])

# splitting data into train and test sets
split_percent = 0.8

X_train = data[:int(data.shape[0]*split_percent), :1000].astype(np.float)
y_train = data[:int(data.shape[0]*split_percent), 1000:1001]
X_test = data[int(data.shape[0]*split_percent): , :1000].astype(np.float)
y_test = data[int(data.shape[0]*split_percent): , 1000:1001]

X_train = X_train.reshape(X_train.shape[0], 1000, 1)
X_test = X_test.reshape(X_test.shape[0], 1000, 1)

# Convolutinal Neural Network model
model = Sequential()
model.add(Conv1D(100, 10, strides=1, input_shape=(1000,1)))
model.add(AveragePooling1D(2))
model.add(Flatten())
model.add(Dense(1000, activation='relu', kernel_regularizer=keras.regularizers.l2(0.02)))
model.add(Dense(12, activation='relu', kernel_regularizer=keras.regularizers.l2(0.01)))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='mean_squared_error', optimizer=keras.optimizers.SGD(lr=0.001), metrics=['accuracy'])

hist = model.fit(X_train, y_train, batch_size=500, epochs=1000)
```

```
model.evaluate(X_test, y_test)

# Results visualization
plt.figure()
plt.title(f'mse vs iteration plot')
plt.xlabel("iteration")
plt.ylabel("mse")
plt.plot(hist.history['loss'], c='b', label='training set loss')
plt.plot(hist.history['val_loss'], c='r', label='testing set loss')
plt.legend(loc='upper right')

plt.figure()
plt.title(f'accuracy vs iteration plot')
plt.xlabel("iteration")
plt.ylabel("accuracy")
plt.plot(hist.history['acc'][::10], c='b', label='training set accuracy')
plt.plot(hist.history['val_acc'][::10], c='r', label='testing set accuracy')
plt.legend(loc='upper left')
```

Results:

1) Conv layer: Filters = 150, filter_size = 700, stride=1
   Fully conncected Layers:
   FC1 = 1000 neurons, FC2 = 16 neurons, output layer = 1 neuron
   L2 regularization in FC1 lambda = 0.01
   Optimizer = adam
   Iteration = 600
   Batch_size = 500
   Test Accuracy = 0.91
   Confusion matrix = [[89 8]
   [10 93]]

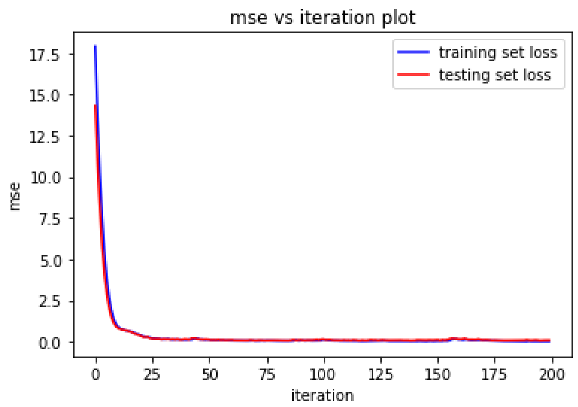


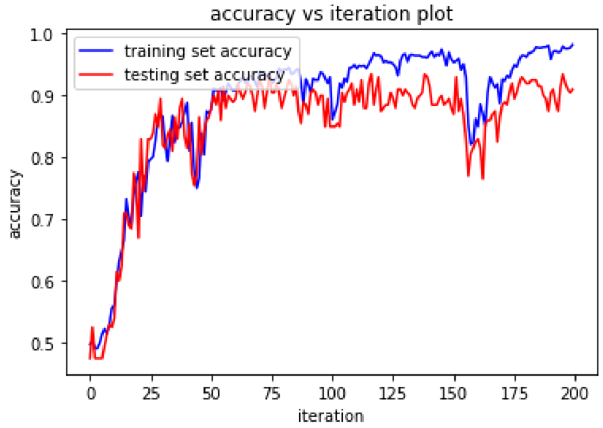


2) Conv_1: Filters = 512 filter_size = 650, stride=1
   Conv_2: Filters = 256, filter_size = 64, stride=1
   Fully conncected Layers:
   FC1 = 1000 neurons, FC2 = 128 neurons, FC3 = 16 neurons, output layer = 1 neuron
   Optimizer = adam
   Iteration = 200

Batch_size = 500
Test Accuracy = 0.91

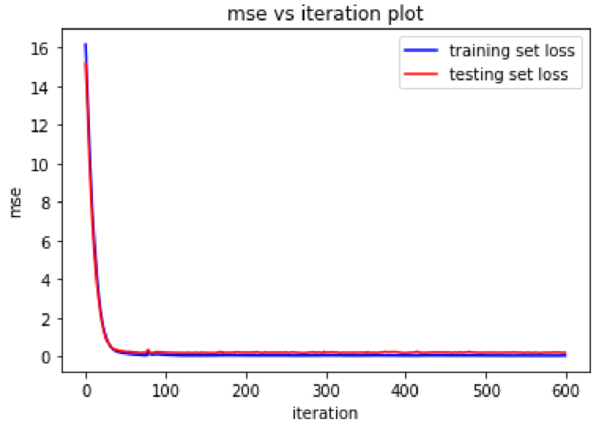
Confusion matrix = [[84 11]
[7 98]]

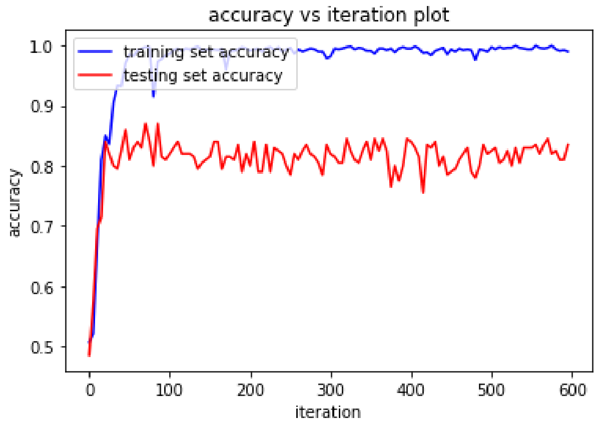




3) Conv_1: Filters = 100, filter_size = 10, stride=1
Conv_2: Filters = 16, filter_size = 10, stride=1
Fully conncected Layers:
FC1 = 1000 neurons, FC2 = 128 neurons, FC3 = 16 neurons, output layer = 1 neuron
Optimizer = adam
Iteration = 600
Batch_size = 1000
Test Accuracy = 0.865

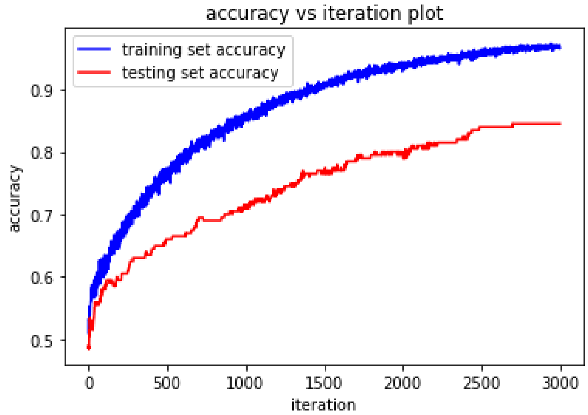
Confusion matrix = [[94 16]
[11 79]]





4) Filters = 100, filter_size = 10, stride=1
Fully conncected Layers = FC1 = 1000 neurons, FC3 = 12 neurons, output layer = 1 neuron
L2 regularization in FC1 lambda = 0.01
Learning rate = 0.001 (sgd optimizer)
Iteration = 3000
Batch_size = 500
Test set Accuracy: 0.845

Confusion matrix = [[91 18]
[13 78]]



2. Convolutional Autoencoder

```
import numpy as np
from preprocessing import NormalScaler
from scipy.io import loadmat
from keras.models import Model, Sequential
from keras.layers import Dense, Conv1D, Flatten, Lambda, MaxPooling1D, UpSampling1D, Conv2DTranspose, Input,Reshape
from keras.engine.topology import Layer
import keras

class Conv1DTranspose(Layer):
    def __init__(self, filters, kernel_size, strides=1, *args, **kwargs):
        self._filters = filters
        self._kernel_size = (1, kernel_size)
        self._strides = (1, strides)
        self._args, self._kwargs = args, kwargs
        super(Conv1DTranspose, self).__init__()

    def build(self, input_shape):
        # print("build", input_shape)
        self._model = Sequential()
        self._model.add(Lambda(lambda x: K.expand_dims(x,axis=1), batch_input_shape=input_shape))
        self._model.add(Conv2DTranspose(self._filters,
                                        kernel_size=self._kernel_size,
                                        strides=self._strides,
                                        *self._args, **self._kwargs))
        self._model.add(Lambda(lambda x: x[:,0]))
        # self._model.summary()
        super(Conv1DTranspose, self).build(input_shape)

    def call(self, x):
        return self._model(x)

    def compute_output_shape(self, input_shape):
        return self._model.compute_output_shape(input_shape)

if __name__=='__main__':
    # data input
    data = loadmat('./data_for_cnn.mat')['ecg_in_window']

    np.random.shuffle(data)
```

```
# data preprocessing
scaler = NormalScaler()
for j in range(data.shape[1]):
    scaler.fit(data[:,j])
    data[:,j] = scaler.transform(data[:,j])

# holdout split
split_percent = 0.7

X_train = data[:int(data.shape[0]*split_percent), :].astype(np.float)
X_test = data[int(data.shape[0]*split_percent): , :].astype(np.float)

X_train = X_train.reshape(X_train.shape[0], 1000, 1)
X_test = X_test.reshape(X_test.shape[0], 1000, 1)

# number of filters
filters = 10

# Encoder
inp = Input(shape=(1000,1))
l1 = Conv1D(filters, 10, strides=2, activation='relu')(inp)
l2 = MaxPooling1D(2)(l1)

l3 = Flatten()(l2)
l4 = Dense(248*filters, activation='relu', kernel_regularizer=keras.regularizers.l2(0.01))(l3)

# Decoder
l4 = Reshape((248, filters)) (l4)
l5 = UpSampling1D(2)(l4)
out = Conv1DTranspose(1, 10, strides=2)(l5)

model = Model(inp, out)

model.compile(loss='mean_squared_error', optimizer='adam')
hist = model.fit(X_train, X_train, validation_data=(X_test,X_test) , batch_size=500, epochs=200)

# Results visualization
plt.figure()
plt.title(f'mse vs iteration plot')
plt.xlabel("iteration")
plt.ylabel("mse")
plt.legend(loc='upper right')
plt.plot(hist.history['val_loss'],c='r',label='validation set loss')
plt.plot(hist.history['loss'],c='b',label='training set loss')
```

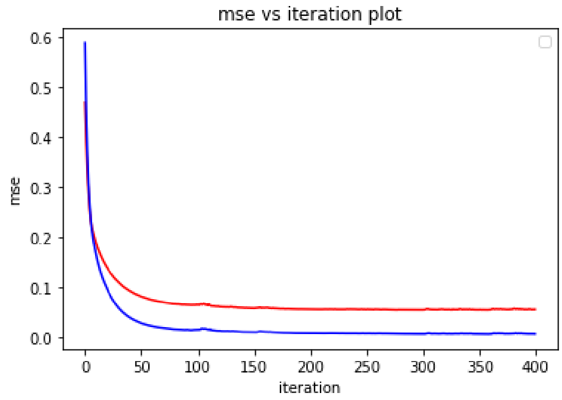
Results:

No of filters = 10   Filter_size = 10

Strides = 2   Epochs = 400

Train set Error: 0.0064   Test set Error: 0.0554

```
Model: "model_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_13 (InputLayer)        (None, 1000, 1)           0
_________________________________________________________________
conv1d_13 (Conv1D)           (None, 496, 10)           110
_________________________________________________________________
max_pooling1d_13 (MaxPooling (None, 248, 10)           0
_________________________________________________________________
flatten_13 (Flatten)         (None, 2480)              0
_________________________________________________________________
dense_12 (Dense)             (None, 2480)              6152880
_________________________________________________________________
reshape_12 (Reshape)         (None, 248, 10)           0
_________________________________________________________________
up_sampling1d_10 (UpSampling (None, 496, 10)           0
_________________________________________________________________
conv1d_transpose_10 (Conv1DT (None, 1000, 1)           0
=================================================================
Total params: 6,152,990
Trainable params: 6,152,990
Non-trainable params: 0
```



3. Neuro Fuzzy Classifier using Linguistic Hedges

```python
import numpy as np
import matplotlib.pyplot as plt
from preprocessing import NormalScaler
import pandas as pd

class Network:
    def __init__(self, X, y, n_rules):
        m = X.shape[0]
        n = X.shape[1]
        k = y.shape[1]
        self.cost_arr = {'train':[], 'test':[]}
        self.att = {
        # input layer
        'in': np.ndarray((n,1)),
        # Layer 1 (Membership Layer)
        'mu': np.ndarray(shape = (n_rules, n)),
        'c': np.random.randn(n_rules, n),
        # 'c': np.random.uniform(low=-1, high=1 , size=(n_rules, n)),
        'c_err': np.zeros(shape=(n_rules, n)),
        'sigma': np.random.rand(n_rules, n),
        'sigma_err': np.zeros(shape = (n_rules, n)),

        # Layer 2 (Power Layer)
        'alpha': np.random.randn(n_rules, n),
        'p': np.random.uniform(low=0.1, high=4, size=(n_rules, n)),
        'p_err': np.zeros(shape = (n_rules, n)),

        # Layer 3 (Fuzzification Layer)
        'beta': np.random.randn(n_rules, 1),

        # Layer 4 (De-fuzzification Layer)
        'o': np.random.randn(1, k),
        'w':np.random.randn(n_rules, k),
```

```python
        'w_err':np.zeros(shape=(n_rules, k)),

        # Layer 5 (Normalization Layer)
        'h':np.ndarray(shape = (1, k)),
        'delta': 1,

        }

    def feed_forward(self, X, j):
        '''
        In this function the given data set samples are propogated
        forward in the neural network.
        '''
        self.att['in'] = X[j].reshape(-1,1)
        self.att['mu'] = np.exp(-0.5 * np.square((self.att['in'].T - self.att['c'])/(self.att['sigma'])))
        self.att['alpha'] = np.power(self.att['mu'], self.att['p'])
        self.att['beta'] = np.product(self.att['alpha'], axis=1).reshape(-1,1)
        self.att['o'] = self.att['beta'].T @ self.att['w']
        self.att['delta'] = np.sum(self.att['o'])
        self.att['h'] = (self.att['o']/self.att['delta'])

        # self.print_shapes()
        return self.att['h']

    def train(self, X, y, X_test, y_test, lr, batch_size, max_iter):
        '''
        This function takes the training data and target values,
        applies forward propogation, then applies backward propogation
        to update the paramater matrices.
        batch gradient descent has been used to update weights.
        '''
        m = y.shape[0]
        k = y.shape[0]
        # self.att['p'].fill(0.0)
        for iteration in range(max_iter):
            for i in range(0,m-batch_size+1,batch_size):
                self.att['c_err'].fill(0)
                self.att['p_err'].fill(0)
                self.att['sigma_err'].fill(0)
                self.att['w_err'].fill(0)
                self.att['b_err'] = 0

                for j in range(i,i+batch_size):
                    # forward propogation
                    self.feed_forward(X, j)

                    # Backpropogation of errors
                    temp = (self.att['h'] - y[j].reshape(1,-1)) * ((1-self.att['h'])/self.att['delta'])
                    temp = ((self.att['beta'] @ temp).T)
                    temp = self.att['w'] @ temp

                    self.att['c_err'] += (temp @ self.att['p']) * (X[j].reshape(1,-
1) - self.att['c'])/(np.square(self.att['sigma']))
```

```
                    self.att['p_err'] += temp @ np.log(abs(self.att['mu']))

                    self.att['sigma_err'] += (temp @ self.att['p']) * (np.square(X[j].reshape(1,-
1) - self.att['c'])/((self.att['sigma'])**3))

                    self.att['w_err'] += self.att['beta'] @ ((self.att['h'] - y[j].reshape(1,-1)) \
                                        * (self.att['delta'] - self.att['o'])/(np.square(self.att['delta'])))


                # updating parameters after backpropogating each batch
                self.att['c'] -= (lr/(batch_size*k))*self.att['c_err']
                self.att['p'] -= (lr/(batch_size*k))*self.att['p_err']
                self.att['sigma'] -= (lr/(batch_size*k))*self.att['sigma_err']
                self.att['w'] -= (lr/(batch_size*k))*self.att['w_err']

            self.cost_arr['train'].append(self.get_cost(X,y))
            self.cost_arr['test'].append(self.get_cost(X_test,y_test))

    def get_cost(self, X, y):
        cost = 0
        for i in range(y.shape[0]):
            # forward propogation
            self.feed_forward(X, i)
            cost += np.sum((self.att['h']-y[i].reshape(1,-1))**2)
        return cost/(2*X.shape[0]*y.shape[1])

    def predict(self, X):
        pred = np.ndarray((X.shape[0],3))
        for i in range(X.shape[0]):
            self.feed_forward(X, i)
            pred[i] = self.att['h']
        return pred

    def evaluate(self, X, y):
        acc = 0
        for i in range(y.shape[0]):
            self.feed_forward(X, i)
            if int(np.argmax(self.att['h']))==int(np.argmax(y[i])):
                acc+=1
        loss = self.get_cost(X, y)
        return {'acc':acc/y.shape[0], 'loss':loss}

if __name__ == "__main__":
    # data input
    data = pd.read_excel("./data4.xlsx",header=None)
    data = data.sample(frac=1).reset_index(drop=True)
    data = data.values

    X = data[:, :7]
    y = data[:,7] - 1

    unique_classes = np.unique(y)
    num_classes = len(unique_classes)
```

```
    # data preprocessing
    mscaler = NormalScaler()
    for j in range(X.shape[1]):
        mscaler.fit(X[j])
        X[j] = mscaler.transform(X[j])

    y_cat = (y==unique_classes[0]).astype('int').reshape(-1,1)
    for i in unique_classes[1:]:
        y_cat = np.concatenate((y_cat,(y==i).astype('int').reshape(-1,1)),axis=1)

    # splitting data using holdout cross validation
    train_percent = 0.7
    X_train = X[:int(train_percent*X.shape[0])]
    y_train = y[:int(train_percent*X.shape[0])]
    y_cat_train = y_cat[:int(train_percent*X.shape[0])]
    X_test = X[int(train_percent*X.shape[0]):]
    y_test = y[int(train_percent*X.shape[0]):]
    y_cat_test = y_cat[int(train_percent*X.shape[0]):]

    alpha = 0.5
    batch_size = 16
    max_iter = 600
    n_rules = 16
    model = Network(X_train, y_cat_train, n_rules)
    model.train(X_train, y_cat_train, X_test, y_cat_test, alpha, batch_size, max_iter)

    print('train: ',model.evaluate(X_train,y_cat_train))
    print('test: ', model.evaluate(X_test,y_cat_test))
    # model.feed_forward(X_train, 0)
    # print("COST: ",model.get_cost(X_train, y_cat_train))
    model.evaluate(X_test, y_cat_test)
    # model.print_shapes()

    plt.figure()
    plt.title(f'Cost Function vs iteration plot alpha={alpha} max_iter={max_iter} batch_size={batch_size}\n n_rules={n
_rules}')
    plt.xlabel("iteration")
    plt.ylabel("cost")
    plt.plot(model.cost_arr['train'],c='c',label='training set avg cost')
    plt.plot(model.cost_arr['test'], c='r',label='testing set avg cost')
    plt.legend(loc='upper right')
    plt.savefig(f"./results/{alpha}_{max_iter}_{batch_size}.png")
    plt.show()
```

Results:

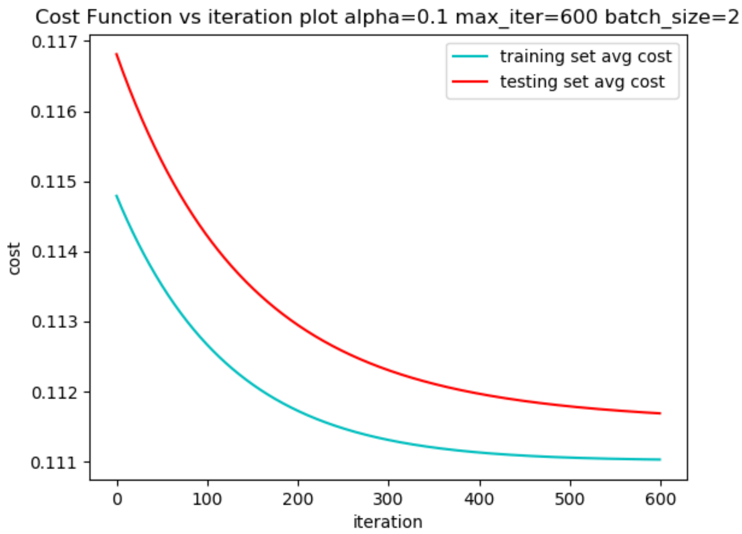
Learning rate = 0.1

No of rules = 10

Epochs = 600 Batch_size = 2

Train loss = 0.11128776886478348 Train accuracy = 0.9875

Test loss = 0.11186907776727173 Test accuracy = 0.8343



Name: S Rohith
Id: 2017A7PS0034H