Appendix

Main.py

```
import numpy as np
import tensorflow as tf
from algorithms.dataloader import Handler
from algorithms.cnn import CNNclassifier


# Global params
NUM_LABELS = 10
BATCH_SIZE = 50

# Add the optional arguments
tf.app.flags.DEFINE_string('train', None,
    'file containing the training data (labels & features).')

tf.app.flags.DEFINE_string('test', None,
    'file containing just the test data.')

tf.app.flags.DEFINE_integer('num_epochs', 1,
    'Number of examples to separate from the training '
    'data for the validation set.')

tf.app.flags.DEFINE_boolean('verbose', False,'Produce verbose output.')

FLAGS = tf.app.flags.FLAGS


def main(argv=None):
    handler = Handler(FLAGS)
    handler.extract_train_and_validation_data(NUM_LABELS)
    handler.extract_test_data()


    # Get the training, validation and testing data
    train_X, train_y = handler.get_train_data()
    val_X, val_y = handler.get_validation_data()
    test_X = handler.get_test_data()


    # Get the shape of the training data.
    train_size = handler.get_train_num_samples()
    num_features = handler.get_num_features()

    # Load model and initialize tensorflow session
    model = CNNclassifier(FLAGS)
    model.form_input_graph(num_features, NUM_LABELS)
    model.load_model()
    model.initialize_session()

    # Fit model
    model.fit(train_X,train_y, val_X, val_y,batch_size=50)

    # Make prediction on test data
    test_prediction = model.predict(test_X)
    # Store results
    outfile = './results/cnn_2d_2layer_full_connected_2layer_tensorflow
                                        .csv'
    handler.save_results(test_prediction, outfile)

if __name__ == '__main__':
    tf.app.run()
```

algorithms/cnn.py

```
import tensorflow.python.platform

import tensorflow as tf
import numpy as np

class CNNclassifier:
    def __init__(self, FLAGS):
        # Get the number of epochs for training.
        self.num_epochs = FLAGS.num_epochs

    def form_input_graph(self, num_features, num_labels):
        # Feed training sample and labels to the graph
        self.num_features = num_features
        self.num_labels = num_labels
        self.x = tf.placeholder("float", shape=[None, num_features])
        self.y_ = tf.placeholder("float", shape=[None,num_labels])

    def load_model(self):
        self.__add_convolutional_layers()
        self.__add_fully_connected_layers()
        self.__add_optimizer()
        self.__add_metrics()


    def __add_convolutional_layers(self):
        ###### 1st layer CNN
        # initialize weight and bias of CNN
        self.W_conv1 = self.weight_variable([5, 5, 1, 32])
        self.b_conv1 = self.bias_variable([32])
        # reshape x to a 4d tensor
        self.x_image = tf.reshape(\
            self.x, [-1,28,self.num_features/28,1])
        # add cnn layer with RELU
        self.h_conv1 = tf.nn.relu(self.conv2d(\
            self.x_image, self.W_conv1) + self.b_conv1)
        # add max pooling
        self.h_pool1 = self.max_pool_2x2(self.h_conv1)

        ###### 2nd Layer CNN
        # initialize weight and bias of CNN
        self.W_conv2 = self.weight_variable([5, 5, 32, 64])
        self.b_conv2 = self.bias_variable([64])
        # add cnn layer with RELU
        self.h_conv2 = tf.nn.relu(self.conv2d(\
            self.h_pool1, self.W_conv2) + self.b_conv2)
        # add max pooling
        self.h_pool2 = self.max_pool_2x2(self.h_conv2)

    def weight_variable(self,shape):
      initial = tf.truncated_normal(shape, stddev=0.1)
      return tf.Variable(initial)

    def bias_variable(self,shape):
      initial = tf.constant(0.1, shape=shape)
      return tf.Variable(initial)

    def conv2d(self, x, W):
      return tf.nn.conv2d(\
        x, W, strides=[1, 1, 1, 1], padding='SAME')

    def max_pool_2x2(self, x):
      return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
        strides=[1, 2, 2, 1], padding='SAME')


    def __add_fully_connected_layers(self):
        ###### Densely Connected layer - 1 (relu)
        self.W_fc1 = self.weight_variable([7 * 7 * 64, 1024])
        self.b_fc1 = self.bias_variable([1024])
        self.h_pool2_flat = tf.reshape(self.h_pool2, [-1, 7*7*64])
        self.h_fc1 = tf.nn.relu(tf.matmul(\
            self.h_pool2_flat, self.W_fc1) + self.b_fc1)

        ###### Dropout layer
        self.keep_prob = tf.placeholder(tf.float32)
        self.h_fc1_drop = tf.nn.dropout(self.h_fc1, self.keep_prob)

        ###### Densely Conected layer -2 (softmax)
        self.W_fc2 = self.weight_variable([1024, self.num_labels])
        self.b_fc2 = self.bias_variable([self.num_labels])
        self.y_conv=tf.nn.softmax(tf.matmul(\
            self.h_fc1_drop, self.W_fc2) + self.b_fc2)


    def __add_optimizer(self):
        # Loss function
        self.cross_entropy = tf.reduce_mean(\
            -tf.reduce_sum(self.y_ * tf.log(self.y_conv),
                reduction_indices=[1]))
        # Optimizer
        self.learning_rate = 5e-4
        self.optimizer = tf.train.AdamOptimizer(\
            self.learning_rate).minimize(self.cross_entropy)

    def __add_metrics(self):
        self.correct_prediction = tf.equal(\
            tf.argmax(self.y_conv,1), tf.argmax(self.y_,1))
        self.accuracy = tf.reduce_mean(tf.cast(\
            self.correct_prediction, tf.float32))
        prediction=tf.argmax(self.y_conv,1)
    def initialize_session(self):
        ## Create and initialize the interactive session
        self.sess = tf.InteractiveSession()
        self.sess.run(tf.initialize_all_variables())


    def fit(self, train_X, train_y, val_X, val_y, batch_size=50):
        train_size = train_X.shape[0]
        # Iterate and train.
        for step in xrange(self.num_epochs * train_size // batch_size):
            offset = (step * batch_size) % train_size
            batch_data = train_X[offset:(offset + batch_size), :]
            batch_labels = train_y[offset:(offset + batch_size)]
            # Train
            self.sess.run(self.optimizer,feed_dict={\
                self.x: batch_data, self.y_: batch_labels, \
                self.keep_prob: 0.5})

            if(step % 10 == 0):
                print 'Step Count:', step
                # Get a validation accuracy
                print 'Validation Acc: ',self.sess.run(self.accuracy, \
                    feed_dict={self.x: val_X, self.y_: val_y, \
                    self.keep_prob: 1.0})

    def predict(self, test_X):
        self.test_prediction = self.sess.run(self.prediction,
                        feed_dict={self.x: test_X, self.keep_prob: 1.0}
                                                            )

        return self.test_prediction
```

algorithms/dataloader.py

```
import numpy as np
import pandas as pd
from sklearn.cross_validation import train_test_split
from sklearn import preprocessing


class Handler:
    def __init__(self, FLAGS):
        # Be verbose?
        self.verbose = FLAGS.verbose
        # Get the data.
        self.train_data_filename = FLAGS.train
        self.test_data_filename = FLAGS.test


    def extract_train_and_validation_data(self,num_labels):
        data = pd.read_csv(self.train_data_filename, header=0).values
        # convert to Numpy array forms
        feature_vec = data[0::,1::]
        labels = data[0::,0]
        # mean normalize features
        min_max_scaler = preprocessing.MinMaxScaler()
        feature_vec = min_max_scaler.fit_transform(feature_vec.T).T

        # convert to one hot form for labels
        labels_onehot = (np.arange(num_labels) == labels[:, None]).
                                            astype(\
            np.float32)

        # divide data into train and validation data
        self.train_X, self.val_X, self.train_y, self.val_y =
                                            train_test_split(\
            feature_vec, labels_onehot, test_size=0.2, random_state=42)


    def extract_test_data(self):
        feature_vec = pd.read_csv(self.test_data_filename, header=0).
                                            values

        # mean normalize features
        min_max_scaler = preprocessing.MinMaxScaler()
        feature_vec = min_max_scaler.fit_transform(feature_vec.T).T

        self.test_X = feature_vec

    def get_train_data(self):
        return self.train_X, self.train_y

    def get_validation_data(self):
        return self.val_X, self.val_y

    def get_test_data(self):
        return self.test_X

    def get_train_num_samples(self):
        return self.train_X.shape[0]

    def get_num_features(self):
        return self.train_X.shape[1]

    def store_results(self, result, outfile):
        # Predict result for test data
        df = pd.DataFrame()
        df['ImageId'] = np.arange(1, self.test_X.shape[0] + 1)
        df['Label'] = result
        df.to_csv(outfile, index=False)
```