Sample Code

1. Demo_e.py

This demo illustrates one of my recent projects using CV and related machine learning method to extract the information from a video clip of a LOL game.

```
import argparse
import cfg
from network import East
from predict import predict
from moviepy.editor import VideoFileClip
from tqdm import tqdm
import torch.utils.data
from torch.autograd import Variable
import os
from crnn import util
from crnn import dataset
from crnn.models import crnn
from crnn import keys
import cv2
import json
import pandas as pd

# This demo shows a recent project using CV to extract information from video clips of LOL and trained with pre-defined model that runs on cuda.

def parse_args():
    """define the working dir"""
    parser = argparse.ArgumentParser()
    parser.add_argument('--path', '-p',
                                default='',
                                help='image path')
    parser.add_argument('--threshold', '-t',
                                default=cfg.pixel_threshold,
                                help='pixel activation threshold')
    return parser.parse_args()
def work(vnum):
    zh = []
    ocr = []
    for fla in tqdm(range(vnum)):
        try:
            cap.set(cv2.CAP_PROP_POS_FRAMES, fla)
```

```
            zh.append(fla)
            ret, frame = cap.read()
            image0 = predict(east_detect, frame, threshold)
            ocrstr = []
            n = 0
            for l, img in image0:
                n += 1
                image = img.convert('L')
                scale = image.size[1] * 1.0 / 32
                w = image.size[0] / scale
                w = int(w)
                # print(w)

                transformer = dataset.resizeNormalize((w, 32))
                image = transformer(image).cuda()
                image = image.view(1, *image.size())
                image = Variable(image)
                model.eval()
                preds = model(image)
                _, preds = preds.max(2)
                preds = preds.squeeze(-2)
                preds = preds.transpose(1, 0).contiguous().view(-1)
                preds_size = Variable(torch.IntTensor([preds.size(0)]))
                raw_pred = converter.decode(preds.data, preds_size.data, raw=True)
                sim_pred = converter.decode(preds.data, preds_size.data, raw=False)
                if len(image0) == 0:
                    ocr.append([fla,0])
                elif len(image0) == 1:
                    ocr.append([fla, all[l, sim_pred]])
                else:
                    ocrstr.append([l, sim_pred])
                    ocr.append([fla, ocrstr])
                # print(dict(zip(zh, ocr)))
                if len(zh) == int(cap.get(7)):
                    break
                    # focr = dict(zip(zh, ocr))
        except:
            pass
        continue

return [ocr]
```

```
if __name__ == '__main__':
    """use pretrained weights and RNN model to run CV on cuda environement """
    args = parse_args()
    img_path = args.path
    threshold = float(args.threshold)

    east = East()
    east_detect = east.east_network()
    east_detect.load_weights(cfg.saved_model_weights_file_path)

    alphabet = keys.alphabet

    converter = util.strLabelConverter(alphabet)
    model = crnn.CRNN(32, 1, len(alphabet) + 1, 256, 1).cuda()
    path = './crnn/samples/netCRNN63.pth'
    model.load_state_dict(torch.load(path))
    videopath = '/home/username/PycharmProjects/videos/'
    finish = []
    for file in os.listdir('./result'):
        if file.endswith('.xlsx'):
            finish.append(file[:-5] + '.mp4')
    flag = 0
    for file in os.listdir(videopath):
        if file not in finish:
            flag += 1
            print('Processing the {} video...'.format(flag))
            size = os.path.getsize(videopath + file)
            if size == 0:
                zong = dict([("Video_Name", file[:-4]), ("Video_Time", 0), ("Total_Frames", 0),("Video_Ocr",
0)])
            else:
                clip = VideoFileClip(videopath + file)
                file_time = clip.duration
                cap = cv2.VideoCapture(videopath + file)
                vnum = int(cap.get(7))
                zong = dict([("Video_Name", file[:-4]), ("Video_Time", file_time), ("Total_Frames", vnum)])
                ocr = work(vnum)
                # focr = dict(zip(zh, ocr))
                print("focr:", ocr)
```

```
            zong["Video_Ocr"] = ocr

        df = pd.DataFrame(zong, index=[0])
        cols = ['Video_Name', 'Video_Time', 'Total_Frames', 'Video_Ocr']
        df = df.ix[:, cols]
        df.to_excel("./result/"+file[:-4]+".xlsx", index=False)
        df.to_hdf("./result/"+file[:-4]+".h5", "data")
        print(df.head())
```

2. Emojify.py

This is a project building a LSTM model to predict the emotion of a sentence and hence add a emoji.

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 31 17:01:19 2019

@author: Dequan
"""
import pandas as pd
import csv
from pandas import read_csv
import numpy as np
# from emo_utils import *
import emoji
import matplotlib.pyplot as plt

#############################################################################################


def read_glove_vecs(glove_file):
    """
    read gloveVec file, initial encoding
    """
    with open(glove_file, 'r', encoding='UTF-8') as f:
        words = set()
        word_to_vec_map = {}
        for line in f:
            line = line.strip().split()
            curr_word = line[0]
            words.add(curr_word)
            word_to_vec_map[curr_word] = np.array(line[1:], dtype=np.float64)
```

```
        i = 1
        words_to_index = {}
        index_to_words = {}
        for w in sorted(words):
            words_to_index[w] = i
            index_to_words[i] = w
            i = i + 1
    return words_to_index, index_to_words, word_to_vec_map


def softmax(x):
    """Compute softmax values for each sets of scores in x."""
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()


def read_csv(filename='data/emojify_data.csv'):
    phrase = []
    emoji = []

    with open(filename) as csvDataFile:
        csvReader = csv.reader(csvDataFile)

        for row in csvReader:
            phrase.append(row[0])
            emoji.append(row[1])

    X = np.asarray(phrase)
    Y = np.asarray(emoji, dtype=int)

    return X, Y


def convert_to_one_hot(Y, C):
    Y = np.eye(C)[Y.reshape(-1)]
    return Y


emoji_dictionary = {"0": ":heart:",        # :heart: prints a black instead of red heart depending on the font
                    "1": ":baseball:",
                    "2": ":smile:",
```

```
                    "3": ":disappointed:",
                    "4": ":fork_and_knife:"}


def label_to_emoji(label):
    """
    Converts a label (int or string) into the corresponding emoji code (string)
    ready to be printed
    """
    return emoji.emojize(emoji_dictionary[str(label)], use_aliases=True)


def print_predictions(X, pred):
    print()
    for i in range(X.shape[0]):
        print(X[i], label_to_emoji(int(pred[i])))


def plot_confusion_matrix(y_actu, y_pred, title='Confusion matrix', cmap=plt.cm.gray_r):
"""
plot the confusion matrix
"""
    df_confusion = pd.crosstab(y_actu, y_pred.reshape(y_pred.shape[0],), rownames=['Actual'],
colnames=['Predicted'], margins=True)

    df_conf_norm = df_confusion / df_confusion.sum(axis=1)

    plt.matshow(df_confusion, cmap=cmap)    # imshow
    # plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(df_confusion.columns))
    plt.xticks(tick_marks, df_confusion.columns, rotation=45)
    plt.yticks(tick_marks, df_confusion.index)
    # plt.tight_layout()
    plt.ylabel(df_confusion.index.name)
    plt.xlabel(df_confusion.columns.name)


def predict(X, Y, W, b, word_to_vec_map):
    """
    Given X (sentences) and Y (emoji indices), predict emojis and compute the
```

```
    accuracy of your model over the given set.

    Arguments:
    X -- input data containing sentences, numpy array of shape (m, None)
    Y -- labels, containing index of the label emoji, numpy array of shape (m, 1)

    Returns:
    pred -- numpy array of shape (m, 1) with your predictions
    """
    m = X.shape[0]
    pred = np.zeros((m, 1))

    for j in range(m):                       # Loop over training examples

        # Split jth test example (sentence) into list of lower case words
        words = X[j].lower().split()

        # Average words' vectors
        avg = np.zeros((50,))
        for w in words:
            avg += word_to_vec_map[w]
        avg = avg/len(words)

        # Forward propagation
        Z = np.dot(W, avg) + b
        A = softmax(Z)
        pred[j] = np.argmax(A)

    print("Accuracy: "+str(np.mean((pred[:] == Y.reshape(Y.shape[0], 1)[:]))))

    return pred
#############################################################################################


X_train, Y_train = read_csv('data/train_emoji.csv')

X_test, Y_test = read_csv('data/tesss.csv')
#############
# test
index = 1
print('test==>', X_train[index], label_to_emoji(Y_train[index]))
```

```
#############

maxLen = len(max(X_train, key=len).split())

Y_oh_train = convert_to_one_hot(Y_train, C=5)
Y_oh_test = convert_to_one_hot(Y_test, C=5)

index = 50
print(Y_train[index], "is converted into one hot", Y_oh_train[index])

word_to_index, index_to_word, word_to_vec_map = read_glove_vecs('data/glove.6B.50d.txt')
# load world-to-vec map


def sentence_to_avg(sentence, word_to_vec_map):
    """
    Converts a sentence (string) into a list of words (strings). Extracts the
    GloVe representation of each word and averages its value into a single vector
    encoding the meaning of the sentence.

    Arguments:
    sentence -- string, one training example from X
    word_to_vec_map -- dictionary mapping every word in a vocabulary into its
    50-dimensional vector representation

    Returns:
    avg -- average vector encoding information about the sentence, numpy-array of shape (50,)
    """


    # Step 1: Split sentence into list of lower case words ( ≈  1 line)
    words = (sentence.lower()).split()

    # Initialize the average word vector, should have the same shape as your word vectors.
    avg = np.zeros((50,))

    # Step 2: average the word vectors. You can loop over the words in the list "words".
    for w in words:
        avg += word_to_vec_map[w]
    avg = avg / len(words)
```

```
    return avg


avg = sentence_to_avg("Morrocan couscous is my favorite dish", word_to_vec_map)
print("avg = ", avg)


def model(X, Y, word_to_vec_map, learning_rate=0.01, num_iterations=400):
    """
    Model to train word vector representations in numpy.

    Arguments:
    X -- input data, numpy array of sentences as strings, of shape (m, 1)
    Y -- labels, numpy array of integers between 0 and 7, numpy-array of shape (m, 1)
    word_to_vec_map -- dictionary mapping every word in a vocabulary into its
    50-dimensional vector representation learning_rate -- learning_rate for the
    stochastic gradient descent algorithm num_iterations -- number of iterations

    Returns:
    pred -- vector of predictions, numpy-array of shape (m, 1)
    W -- weight matrix of the softmax layer, of shape (n_y, n_h)
    b -- bias of the softmax layer, of shape (n_y,)
    """

    np.random.seed(1)

    # Define number of training examples
    m = Y.shape[0]                                  # number of training examples
    n_y = 5                                         # number of classes
    n_h = 50                                        # dimensions of the GloVe vectors

    # Initialize parameters using Xavier initialization
    W = np.random.randn(n_y, n_h) / np.sqrt(n_h)
    b = np.zeros((n_y,))

    # Convert Y to Y_onehot with n_y classes
    Y_oh = convert_to_one_hot(Y, C=n_y)

    # Optimization loop
    for t in range(num_iterations):                 # Loop over the number of iterations
        for i in range(m):                          # Loop over the training examples
```

```
            # Average the word vectors of the words from the i'th training example
            avg = sentence_to_avg(X[i], word_to_vec_map)

            # Forward propagate the avg through the softmax layer
            z = np.dot(W, avg) + b
            a = softmax(z)

            # Compute cost using the i'th training label's one hot
            # representation and "A" (the output of the softmax)
            cost = - np.sum(Y_oh[i] * np.log(a))

            # Compute gradients
            dz = a - Y_oh[i]
            dW = np.dot(dz.reshape(n_y, 1), avg.reshape(1, n_h))
            db = dz

            # Update parameters with Stochastic Gradient Descent
            W = W - learning_rate * dW
            b = b - learning_rate * db

        if t % 100 == 0:
            print("Epoch: " + str(t) + " --- cost = " + str(cost))
            pred = predict(X, Y, W, b, word_to_vec_map)

    return pred, W, b


print('X_train.shape', X_train.shape)
print('Y_train.shape', Y_train.shape)
print(np.eye(5)[Y_train.reshape(-1)].shape)
print(X_train[0])
print(type(X_train))

Y = np.asarray([5, 0, 0, 5, 4, 4, 4, 6, 6, 4, 1, 1, 5, 6, 6, 3, 6, 3, 4, 4])
print(Y.shape)

X = np.asarray(['I am going to the bar tonight', 'I love you', 'miss you my dear',
  'Lets go party and drinks','Congrats on the new job','Congratulations',
  'I am so happy for you', 'Why are you feeling bad', 'What is wrong with you',
  'You totally deserve this prize', 'Let us go play football',
```

```
  'Are you down for football this afternoon', 'Work hard play harder',
  'It is suprising how people can be dumb sometimes',
  'I am very disappointed','It is the best day in my life',
  'I think I will end up alone','My life is so boring','Good job',
  'Great so awesome'])

print(X.shape)
print(np.eye(5)[Y_train.reshape(-1)].shape)
print(type(X_train))

pred, W, b = model(X_train, Y_train, word_to_vec_map)
print(pred)

print("Training set:")
pred_train = predict(X_train, Y_train, W, b, word_to_vec_map)
print('Test set:')
pred_test = predict(X_test, Y_test, W, b, word_to_vec_map)

X_my_sentences = np.array(["i adore you", "i love you", "funny lol",
"lets play with a ball", "food is ready", "not feeling happy"])
Y_my_labels = np.array([[0], [0], [2], [1], [4], [3]])

pred = predict(X_my_sentences, Y_my_labels, W, b, word_to_vec_map)
print_predictions(X_my_sentences, pred)

print(Y_test.shape)
print('                    '+ label_to_emoji(0)+ '       ' + label_to_emoji(1) + '       ' +    label_to_emoji(2)+ '       ' +
label_to_emoji(3)+'      ' + label_to_emoji(4))
print(pd.crosstab(Y_test, pred_test.reshape(56,), rownames=['Actual'], colnames=['Predicted'], margins=True))
plot_confusion_matrix(Y_test, pred_test)
########################################################################################
# Emojify V2 this version add keras training method using LSTM

import numpy as np
np.random.seed(0)
from keras.models import Model
from keras.layers import Dense, Input, Dropout, LSTM, Activation
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
from keras.initializers import glorot_uniform
np.random.seed(1)
```

```
def sentences_to_indices(X, word_to_index, max_len):
    """
    Converts an array of sentences (strings) into an array of indices corresponding to words in the sentences.
    The output shape should be such that it can be given to `Embedding()` (described in Figure 4).

    Arguments:
    X -- array of sentences (strings), of shape (m, 1)
    word_to_index -- a dictionary containing the each word mapped to its index
    max_len -- maximum number of words in a sentence. You can assume every sentence in X is no longer than this.

    Returns:
    X_indices -- array of indices corresponding to words in the sentences from X, of shape (m, max_len)
    """

    m = X.shape[0]                                   # number of training examples

    # Initialize X_indices as a numpy matrix of zeros and the correct shape (≈ 1 line)
    X_indices = np.zeros((m, max_len))

    for i in range(m):                               # loop over training examples

        # Convert the ith training sentence in lower case and split is into words.
        # You should get a list of words.
        sentence_words = X[i].lower().split()

        # Initialize j to 0
        j = 0

        # Loop over the words of sentence_words
        for w in sentence_words:
            # Set the (i,j)th entry of X_indices to the index of the correct word.
            X_indices[i, j] = word_to_index[w]
            # Increment j to j + 1
            j = j + 1

    return X_indices
```

```
def pretrained_embedding_layer(word_to_vec_map, word_to_index):
    """
    Creates a Keras Embedding() layer and loads in pre-trained GloVe 50-dimensional vectors.

    Arguments:
    word_to_vec_map -- dictionary mapping words to their GloVe vector representation.
    word_to_index -- dictionary mapping from words to their indices in the vocabulary (400,001 words)

    Returns:
    embedding_layer -- pretrained layer Keras instance
    """

    vocab_len = len(word_to_index) + 1                  # adding 1 to fit Keras embedding (requirement)
    emb_dim = word_to_vec_map["cucumber"].shape[0]      # define dimensionality of your GloVe word vectors (= 50)


    # Initialize the embedding matrix as a numpy array of zeros of shape
    # (vocab_len, dimensions of word vectors = emb_dim)
    emb_matrix = np.zeros((vocab_len, emb_dim))

    # Set each row "index" of the embedding matrix to be the word vector
    # representation of the "index"th word of the vocabulary
    for word, index in word_to_index.items():
        emb_matrix[index, :] = word_to_vec_map[word]

    # Define Keras embedding layer with the correct output/input sizes, make it trainable.
    # Use Embedding(...). Set trainable=False.
    embedding_layer = Embedding(vocab_len, emb_dim)

    # Build the embedding layer, it is required before setting the weights of
    # the embedding layer.
    embedding_layer.build((None,))

    # Set the weights of the embedding layer to the embedding matrix. Your layer is now pretrained.
    embedding_layer.set_weights([emb_matrix])

    return embedding_layer


embedding_layer = pretrained_embedding_layer(word_to_vec_map, word_to_index)
```

```
print("weights[0][1][3] =", embedding_layer.get_weights()[0][1][3])


def Emojify_V2(input_shape, word_to_vec_map, word_to_index):
    """
    Function creating the Emojify-v2 model's graph.

    Arguments:
    input_shape -- shape of the input, usually (max_len,)
    word_to_vec_map -- dictionary mapping every word in a vocabulary into its
    50-dimensional vector representation word_to_index -- dictionary mapping from
    words to their indices in the vocabulary (400,001 words)

    Returns:
    model -- a model instance in Keras
    """

    # Define sentence_indices as the input of the graph,
    # it should be of shape input_shape and dtype 'int32' (as it contains indices).
    sentence_indices = Input(shape=input_shape, dtype=np.int32)

    # Create the embedding layer pretrained with GloVe Vectors (≈1 line)
    embedding_layer = pretrained_embedding_layer(word_to_vec_map, word_to_index)

    # Propagate sentence_indices through your embedding layer, you get back the embeddings
    embeddings = embedding_layer(sentence_indices)

    # Propagate the embeddings through an LSTM layer with 128-dimensional hidden state
    # Be careful, the returned output should be a batch of sequences.
    X = LSTM(128, return_sequences=True)(embeddings)
    # Add dropout with a probability of 0.5
    X = Dropout(0.5)(X)
    # Propagate X trough another LSTM layer with 128-dimensional hidden state
    # Be careful, the returned output should be a single hidden state, not a batch of sequences.
    X = LSTM(128)(X)
    # Add dropout with a probability of 0.5
    X = Dropout(0.5)(X)
    # Propagate X through a Dense layer with softmax activation to get back a batch of 5-dimensional vectors.
    X = Dense(5, activation='softmax')(X)
    # Add a softmax activation
    X = Activation('softmax')(X)
```

```
    # Create Model instance which converts sentence_indices into X.
    model = Model(sentence_indices, X)

    return model


model = Emojify_V2((maxLen,), word_to_vec_map, word_to_index)
model.summary()

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
X_train_indices = sentences_to_indices(X_train, word_to_index, maxLen)
Y_train_oh = convert_to_one_hot(Y_train, C=5)

model.fit(X_train_indices, Y_train_oh, epochs=50, batch_size=32, shuffle=True)

X_test_indices = sentences_to_indices(X_test, word_to_index, max_len=maxLen)
Y_test_oh = convert_to_one_hot(Y_test, C=5)
loss, acc = model.evaluate(X_test_indices, Y_test_oh)
print()
print("Test accuracy = ", acc)

# Visulization of the mislabelled examples
C = 5
y_test_oh = np.eye(C)[Y_test.reshape(-1)]
X_test_indices = sentences_to_indices(X_test, word_to_index, maxLen)
pred = model.predict(X_test_indices)
for i in range(len(X_test)):
    x = X_test_indices
    num = np.argmax(pred[i])
    if(num != Y_test[i]):
        print('Expected emoji:'+label_to_emoji(Y_test[i]) +' prediction: '+X_test[i] +
label_to_emoji(num).strip())

# Change the sentence below to see your prediction. Make sure all the words are in the Glove embeddings.
x_test = np.array(['not feeling happy'])
X_test_indices = sentences_to_indices(x_test, word_to_index, maxLen)
print(x_test[0] + ' '+label_to_emoji(np.argmax(model.predict(X_test_indices))))
```