image

May 10, 2021

1 CNN Image Sentiment Analysis

Code to train a CNN image classifier.

Note that the final version of this code was run on the server as my computer was not powerful enough, and hence the latest version of my image classifier is found in server_code/classify_image.py

[1]:
```
import matplotlib
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.models import load_model
from tensorflow.keras.optimizers import SGD
import tensorflow as tf
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from imutils import paths
import random
import matplotlib.pyplot as plt
import numpy as np
import argparse
import pickle
import cv2
import os
import utils.data
import utils.model
from keras_vggface.vggface import VGGFace
import gc
```

[ ]:
```
import wandb
from wandb.keras import WandbCallback
```

```
wandb.login()
```

Set out the parameters of this training run

[3]:
```
# One of ravdess, ravdess-faces, fer+
dataset = "fer+"
FOUR_EMOTIONS = False

# One of RN50
EXPERIMENT = "RN50"

DIMS = (197,197)
```

We use transfer learning, on top of the ResNet CNN, using frames extracted from our videos to get the specific model.

[ ]:
```
if EXPERIMENT in ['RN18-FER+', 'RN18-MS']:
    channels_first = True
else:
    channels_first = False

trainX, valX, testX, trainY, valY, testY, lb = utils.data.
 ↪load_img_dataset(dataset, channels_first, FOUR_EMOTIONS, dims=DIMS)

print(trainX.shape)

# Randomly change the train set so results are more generalizable
if EXPERIMENT in ['RN18-FER+', 'RN18-MS']:
    data_format = 'channels_first'
    train_augmentation = ImageDataGenerator(
        fill_mode="nearest",
        data_format=data_format)
else:
    data_format = 'channels_last'
    train_augmentation = ImageDataGenerator(
        rotation_range=30,
        zoom_range=0.15,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.15,
        horizontal_flip=True,
        fill_mode="nearest",
        data_format=data_format)

val_augmentation = ImageDataGenerator(data_format=data_format)
mean = np.array([123.68, 116.779, 103.939], dtype="float32")
train_augmentation.mean = mean
```

```
val_augmentation.mean = mean
```

[9]:
```
def get_model():
    return utils.model.get_model(EXPERIMENT, len(lb.classes_))
```

Set up the training procedure, using stochastic gradient descent. In the server version of this code we use a more complicated training procedure, where we first train the last few layers of the network, and then continue to train the whole network.

[10]:
```
class MyCustomCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        gc.collect()

def train():
    # default hyperparameters
    config_defaults = {
        'batch_size' : 32,
        'learning_rate' : 0.0008475,
        'epochs': 30,
        'momentum' : 0.9,
        'decay': 1e-4
    }

    wandb.init(project='sentiment', config=config_defaults)
    config = wandb.config

    config.architecture_name = EXPERIMENT
    config.dataset = dataset

    # Compile the model, using stochastic gradient descent optimization.
    opt = SGD(lr=config.learning_rate, momentum=config.momentum, decay=config.
 ↪decay / config.epochs)
    model = get_model()
    model.compile(loss="categorical_crossentropy", optimizer=opt,
        metrics=["accuracy"])

    # Now we can start training!
    H = model.fit(
        x = trainX,
        y = trainY,
        batch_size=config.batch_size,
        steps_per_epoch = len(trainX) // config.batch_size,
        validation_data = (valX, valY),
        validation_steps = len(valX) // config.batch_size,
        epochs = config.epochs,
        callbacks = [WandbCallback()]
    )
```

```
    return model
```

Next, we setup a sweep of hyperparameters, using Bayesian optimization, implemented by weights and biases (wandb.ai).

```
[ ]: sweep_config = {
         "method": "bayes",
         "metric": {
             "name": "val_loss",
             "goal": "minimize"
         },
         "parameters":{
             "epochs": {
                 "distribution": "int_uniform",
                 "min": 20,
                 "max": 40
             },
             "batch_size": {
                 "distribution": "int_uniform",
                 "min": 30,
                 "max": 64
             },
             "learning_rate": {
                 "distribution": "uniform",
                 "min": 0.00001,
                 "max": 0.001
             },
             "momentum": {
                 "distribution": "uniform",
                 "min": 0.9,
                 "max": 0.99
             },
             "decay": {
                 "distribution": "uniform",
                 "min": 1e-6,
                 "max": 1e-2
             }
         },
         "early_terminate" :
         {
             "type": "hyperband",
             "min_iter": 3
         }
     }

     wandb.sweep(sweep_config, project='sentiment')
```

```
wandb.agent(sweep_id, project='sentiment', function=train)
```