brucechou1983 / CheXNet-Keras Public

Code Issues 14 Pull requests 5 Actions Projects Wiki Security Insights

master

CheXNet-Keras / train.py / <> Jump to

brucechou1983 Cleanup

1 contributor

228 lines (203 sloc) | 8.39 KB

```python
import json
import shutil
import os
import pickle
from callback import MultipleClassAUROC, MultiGPUModelCheckpoint
from configparser import ConfigParser
from generator import AugmentedImageSequence
from keras.callbacks import ModelCheckpoint, TensorBoard, ReduceLROnPlateau
from keras.optimizers import Adam
from keras.utils import multi_gpu_model
from models.keras import ModelFactory
from utility import get_sample_counts
from weights import get_class_weights
from augmenter import augmenter


def main():
    # parser config
    config_file = "./config.ini"
    cp = ConfigParser()
    cp.read(config_file)

    # default config
    output_dir = cp["DEFAULT"].get("output_dir")
    image_source_dir = cp["DEFAULT"].get("image_source_dir")
    base_model_name = cp["DEFAULT"].get("base_model_name")
    class_names = cp["DEFAULT"].get("class_names").split(",")

    # train config
    use_base_model_weights = cp["TRAIN"].getboolean("use_base_model_weights")
    use_trained_model_weights = cp["TRAIN"].getboolean("use_trained_model_weights")
    use_best_weights = cp["TRAIN"].getboolean("use_best_weights")
    output_weights_name = cp["TRAIN"].get("output_weights_name")
    epochs = cp["TRAIN"].getint("epochs")
    batch_size = cp["TRAIN"].getint("batch_size")
    initial_learning_rate = cp["TRAIN"].getfloat("initial_learning_rate")
    generator_workers = cp["TRAIN"].getint("generator_workers")
    image_dimension = cp["TRAIN"].getint("image_dimension")
    train_steps = cp["TRAIN"].get("train_steps")
    patience_reduce_lr = cp["TRAIN"].getint("patience_reduce_lr")
    min_lr = cp["TRAIN"].getfloat("min_lr")
    validation_steps = cp["TRAIN"].get("validation_steps")
    positive_weights_multiply = cp["TRAIN"].getfloat("positive_weights_multiply")
    dataset_csv_dir = cp["TRAIN"].get("dataset_csv_dir")
    # if previously trained weights is used, never re-split
    if use_trained_model_weights:
        # resuming mode
        print("** use trained model weights **")
        # load training status for resuming
        training_stats_file = os.path.join(output_dir, ".training_stats.json")
        if os.path.isfile(training_stats_file):
            # TODO: add loading previous learning rate?
            training_stats = json.load(open(training_stats_file))
        else:
            training_stats = {}
    else:
        # start over
        training_stats = {}

    show_model_summary = cp["TRAIN"].getboolean("show_model_summary")
    # end parser config

    # check output_dir, create it if not exists
    if not os.path.isdir(output_dir):
        os.makedirs(output_dir)

    running_flag_file = os.path.join(output_dir, ".training.lock")
    if os.path.isfile(running_flag_file):
        raise RuntimeError("A process is running in this directory!!!")
    else:
        open(running_flag_file, "a").close()

    try:
        print(f"backup config file to {output_dir}")
        shutil.copy(config_file, os.path.join(output_dir, os.path.split(config_file)[1]))

        datasets = ["train", "dev", "test"]
        for dataset in datasets:
            shutil.copy(os.path.join(dataset_csv_dir, f"{dataset}.csv"), output_dir)

        # get train/dev sample counts
        train_counts, train_pos_counts = get_sample_counts(output_dir, "train", class_names)
        dev_counts, _ = get_sample_counts(output_dir, "dev", class_names)

```

```python
    # compute steps
    if train_steps == "auto":
        train_steps = int(train_counts / batch_size)
    else:
        try:
            train_steps = int(train_steps)
        except ValueError:
            raise ValueError(f"""
            train_steps: {train_steps} is invalid,
            please use 'auto' or integer.
            """)
    print(f"** train_steps: {train_steps} **")

    if validation_steps == "auto":
        validation_steps = int(dev_counts / batch_size)
    else:
        try:
            validation_steps = int(validation_steps)
        except ValueError:
            raise ValueError(f"""
            validation_steps: {validation_steps} is invalid,
            please use 'auto' or integer.
            """)
    print(f"** validation_steps: {validation_steps} **")

    # compute class weights
    print("** compute class weights from training data **")
    class_weights = get_class_weights(
        train_counts,
        train_pos_counts,
        multiply=positive_weights_multiply,
    )
    print("** class_weights **")
    print(class_weights)

    print("** load model **")
    if use_trained_model_weights:
        if use_best_weights:
            model_weights_file = os.path.join(output_dir, f"best_{output_weights_name}")
        else:
            model_weights_file = os.path.join(output_dir, output_weights_name)
    else:
        model_weights_file = None

    model_factory = ModelFactory()
    model = model_factory.get_model(
        class_names,
        model_name=base_model_name,
        use_base_weights=use_base_model_weights,
        weights_path=model_weights_file,
        input_shape=(image_dimension, image_dimension, 3))

    if show_model_summary:
        print(model.summary())

    print("** create image generators **")
    train_sequence = AugmentedImageSequence(
        dataset_csv_file=os.path.join(output_dir, "train.csv"),
        class_names=class_names,
        source_image_dir=image_source_dir,
        batch_size=batch_size,
        target_size=(image_dimension, image_dimension),
        augmenter=augmenter,
        steps=train_steps,
    )
    validation_sequence = AugmentedImageSequence(
        dataset_csv_file=os.path.join(output_dir, "dev.csv"),
        class_names=class_names,
        source_image_dir=image_source_dir,
        batch_size=batch_size,
        target_size=(image_dimension, image_dimension),
        augmenter=augmenter,
        steps=validation_steps,
        shuffle_on_epoch_end=False,
    )

    output_weights_path = os.path.join(output_dir, output_weights_name)
    print(f"** set output weights path to: {output_weights_path} **")

    print("** check multiple gpu availability **")
    gpus = len(os.getenv("CUDA_VISIBLE_DEVICES", "1").split(","))
    if gpus > 1:
        print(f"** multi_gpu_model is used! gpus={gpus} **")
        model_train = multi_gpu_model(model, gpus)
        # FIXME: currently (Keras 2.1.2) checkpoint doesn't work with multi_gpu_model
        checkpoint = MultiGPUModelCheckpoint(
            filepath=output_weights_path,
            base_model=model,
        )
    else:
        model_train = model
        checkpoint = ModelCheckpoint(
             output_weights_path,
             save_weights_only=True,
             save_best_only=True,
             verbose=1,
        )

    print("** compile model with class weights **")
    optimizer = Adam(lr=initial_learning_rate)
    model_train.compile(optimizer=optimizer, loss="binary_crossentropy")
    auroc = MultipleClassAUROC(
        sequence=validation_sequence,
        class_names=class_names,
            weights_path=output_weights_path,
            stats=training_stats,
            workers=generator_workers,
        )
        callbacks = [
            checkpoint,
            TensorBoard(log_dir=os.path.join(output_dir, "logs"), batch_size=batch_size),
            ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=patience_reduce_lr,
                              verbose=1, mode="min", min_lr=min_lr),
            auroc,
        ]

        print("** start training **")
        history = model_train.fit_generator(
            generator=train_sequence,
            steps_per_epoch=train_steps,
            epochs=epochs,
            validation_data=validation_sequence,
            validation_steps=validation_steps,
            callbacks=callbacks,
            class_weight=class_weights,
            workers=generator_workers,
            shuffle=False,
        )

        # dump history
        print("** dump history **")
        with open(os.path.join(output_dir, "history.pkl"), "wb") as f:
            pickle.dump({
                "history": history.history,
                "auroc": auroc.aurocs,
            }, f)
        print("** done! **")

    finally:
        os.remove(running_flag_file)


if __name__ == "__main__":
    main()
```