

Model Composition of Neural Networks

This repository contains code to reproduce the results of our paper. It also contains additional code/configs for other experiments and datasets we used, which have not necessarily appeared in the paper.

For the object detection task please refer to: [Object Detection](#)

For the image classification task please refer to: [Image Classification](#)

Model Composition for Object Detection

Pre-requisites

- Tensorflow 2.1

Setting-up locally

1. Create conda environment and install dependencies

```
conda env create -f detection.yml
```

Setting-up on cloud

1. Create custom image using *Dockerfile*.
Required files for creating custom image:
 - [openmpi-4.0.0.tar.gz](#)
 - [tensorflow_gpu-2.1.0-cp36-cp36m-manylinux2010_x86_64.whl](#)
2. Set the **--cloud** parameter to 'True' in *train.py*, *labeler.py*, *ensemble.py*, and *main.py*

How to Run

-- To run model-composition, we use the step-by-step scripts. -- All the examples below except the last four are commands for running locally. Simply add *--cloud* in the command, and change all the local path to S3 path to enable running on cloud.

We need 4 scripts to run model-composition in 7 steps: *labeler.py*, *ensemble.py*, *dataset/create_pseudo_labels_tfrecord_from_json.py*, and *main.py* in *efficientdet* folder.

- Step 1: *labeler.py* to generate pseudo-labels with the baseline models
- Step 2: *ensemble.py* to compose the pseudo-labels of several models
- Step 3: *dataset/create_pseudo_labels_tfrecord_from_json.py* to generate tfrecord of pseudo-labels
- Step 4: *main.py* to train the composed model from the tfrecord
- Step 5: *main.py* to evaluate the composed model
- Step 6: *main.py* to finetune the composed model

- Step 7: main.py to evaluate the finetuned model

Step 1: generate pseudo-labels with the baseline models

1.1 Example of running simple labeler.py:

The following command will generate pseudo-labels for COCO/train data with the best checkpoint of the COCO baseline model.

```
python labeler.py --model_path /home/myuser/checkpoints/best-ckpt/COCO
                  --model_name efficientdet-d0
                  --data_path /data/myuser/coco2017/train2017
                  --output_dir /data/myuser/pseudo_labels/COCO
                  --template_dir /home/myuser
                  --gpu 0
```

INPUTS:

- `--model_path` is the initial checkpoint used to generate pseudo-labels.
- `--model_name(optional)` is the name of the model, can be one of `efficientdet-d0`, `efficientdet-d1`, `efficientdet-d2`, `efficientdet-d3`, `efficientdet-d4`, `efficientdet-d5`, `efficientdet-d6`, `efficientdet-d7`, and `retinanet-50`.
- `--data_path` is the path to raw image files.
- Output pseudo-labels will be saved to `--output_dir`.
- Note that `--template_dir` is the **directory** that contains the template of the output xml annotation. An example of the template is `template.xml` in `efficientdet/annotations` folder.
- `--gpu(optional)` is the gpu index to use.
- We don't need to specify `--hparams` here because all the default hyperparameters in `hparams_config.py` work for COCO data.

OUTPUTS:

Pascal format of xml annotation files

1.2 Example of running labeler.py with change of class id:

Below is another example of generating pseudo-labels, but here we want the pseudo-labels to have different class id with the baseline.

```
python labeler.py --model_path /home/myuser/checkpoints/best-ckpt/voc
```

```

--model_name efficientdet-d0
--data_path /data/myuser/coco2017/train2017
--output_dir
/data/myuser/pseudo_labels/scen_3_1/voc_on_cocovoc_union
--template_dir /home/myuser
--labels_file
/home/myuser/label_maps/pascal_map_to_COCO.txt
--gpu 0
--hparams "num_classes=20"

```

- We enable the change of class id by using the map in *--labels_file*, which maps the output id of the model to a new desired id. An example of the labels_file is *efficientdet/label_files/pascal_map_to_coco.txt* which maps VOC id to COCO id. This is useful when baseline models and the composed model have different class ids. For example, in our scenario 3, one baseline is trained with VOC data whose class id is from 1 to 20, however the composed model has class id from 1 to 90; we need to map the pseudo-labels of the VOC baseline to the proper id between [1,90].
- And since the VOC model has 20 classes, we use *--hparams* to overwrite the default number of classes.

Step 2: Combined all pseudo-labels into one json file

We provide 4 methods of combining pseudo-labels: **affirmative**, **consensus**, **unanimous**, and **partial_unanimous**.

- affirmative: All the pseudo-labels are kept. This means that whenever one of the baseline models that produce

the initial predictions says that a region contains an object, such a detection is considered as valid.

- consensus: Only the prediction which length is greater than $m/2$ (where m is the number of baseline models) are kept.

This means that the majority of the initial checkpoints must agree to consider that a region contains an object.

- unanimous: Only the prediction which length is equal to the number of baseline models are kept. This means that all

the initial checkpoints must agree to consider that a region contains an object.

- partial_unanimous: In the case, we use unanimous for overlapping classes of baseline models, and use affirmative for the

remaining non-overlapping classes. For example, if baseline models are trained from VOC and COCO respectively.

We use unanimous for 20 classes in VOC which overlaps with the same 20 classes in COCO, and use affirmative for the remaining 60 classes in COCO.

2.1 Example of running ensemble.py with the affirmative option:

Below is an example of combining pseudo-labels of 2 baseline models of our scenario 3.

```
python ensemble.py --xml_dir_list
/data/myuser/pseudo_labels/scen_3/COCO_on_cocovoc_union

/data/myuser/pseudo_labels/scen_3/voc_on_cocovoc_union
--data_dir /data/myuser/coco2017/train2017
--output_dir
/home/myuser/ensembled/scenario_3_1/affirmative
--labels_file /home/myuser/label_maps/COCO_label_map.txt
--option affirmative
--num_models 2
```

INPUTS:

- `--xml_dir_list` is a list of paths to pseudo-labels generated in Step 1.
- `--data_dir` is the directory of raw images that are used for pseudo-labeling, which should be the same as `--data_path` in Step 1.
- `--labels_file` is different from the one in Step 1, it is the map from class id to class name for the composed model and is required. An example can be found in `*efficientdet/label_files/scenario_1*`: `coco_00_map.txt`, `coco_01_map.txt`, `coco_02_map.txt` are for `labeler.py`, and `coco_union_label_map.txt` is for `ensemble.py`; `coco_union_label_map.txt` have the same ids as the mapped ids in `coco_00_map.txt`, `coco_01_map.txt`, and `coco_02_map.txt`
- `--num_models` is the number of models to compose.
- `--option(optional)` specifies the method to use for composing pseudo-labels; should be one of 'affirmative', 'consensus', 'unanimous', and 'partial_unanimous'.

OUTPUTS:

A json file named `ensembled.json` under the `output_dir` (the json file will follow the COCO json format)

Some example ensembled files are

2.2 Example of running ensemble.py with the partial_unanimous option:

-- Commands of running the affirmative, consensus, and unanimous methods share similar commands like 2.1 *Example* by only changing the *--option* variable; But running *partial_unanimous* requires to specify *--overlap_classes* which is the list of overlapping class ids shared among baseline models.

-- Below is an example of applying *partial_unanimous* method to our scenario 1. The overlapping classes of 3 baseline models are 'person' and 'bicycle' classes whose class ids are 0 and 1 respectively.

```
python ensemble.py --xml_dir_list
/data/myuser/pseudo_labels/scen_1/coco_00_on_union

/data/myuser/pseudo_labels/scen_1/coco_01_on_union

/data/myuser/pseudo_labels/scen_1/coco_02_on_union
--data_dir
/data/myuser/object_detection/scenario_1/union/train
--output_dir
/home/myuser/ensembled/scenario_1/partial_unanimous
--labels_file
/home/myuser/label_maps/scenario_1/coco_union_label_map.txt
--option partial_unanimous
--overlap_classes 0 1
--num_models 3
```

Step 3: Generate tfrecord for composed pseudo-labels

We then generate tfrecord from the json file of Step 2 to feed into our training pipeline. Below is one example.

```
python create_pseudo_labels_tfrecord_from_json.py --output_path
/data/myuser/scenario_3/tfrecord
--annotations_file /home/my
user/ensembled/scenario_3_1/affimative/ensembled.json
```

INPUTS:

- *--output_path* is the output directory of the tfrecord.
- *--annotations_file* is the path of the composed json file generated in Step 2.

OUTPUTS:

Tfrecord files

Get the correct anchor (optional)

Before training any models, first use *kmeans_anchors_ratios.py* to get the correct anchor of a dataset if dataset is not COCO. See *kmeans_anchors_ratios.py* for more details.

Here is an example of getting the anchors for Pascal VOC.

```
python kmeans_anchors_ratios.py --instance /data/myuser/PascalVOC/VOC-
tfrecord/json_trainval.json
--input-size 512
--normalizes-bboxes True
--num-runs 3
--num-anchors-ratios 3
--max-iter 300
--min-size 0
--iou-threshold 0.5
--decimals 1
--default-anchors-ratios
[(1.0,1.0),(1.4,0.7),(0.7,1.4)]
--anchors-sizes 32 64 128 256 512
```

Step 4 & 5: Train and evaluate the composed model

We use the same main.py script to do training and evaluation:

INPUTS:

- *--strategy* should always be horovod.
- *--mode* should be 'train' for training, 'eval_all' for evaluation. Note that 'eval_all' will evaluate all the produced checkpoints and output the best checkpoints to the *archive* folder in the *--save_dir*.
- *--training_file_pattern* is only for training and *--validation_file_pattern* is only for evaluation.
- *--backbone_ckpt* is only for train to load the backbone.
- *--eval_samples* and *--eval_batch_size* are only for evaluation.
- *--save_dir*, *--hparams(optional)*, *--train_batch_size(optional)*, *--num_examples_per_epoch*, and *--num_epochs(optional)* are the same for train and evaluation if train and evaluation are for the same model.

$$\text{num_examples_per_epoch} = \text{ceil}\left(\frac{\text{number of samples in train dataset}}{\text{number of gpus used for training}}\right)$$

OUTPUTS:

All checkpoints and timeline json files

Example of training on cloud:

The following command shows how to train an efficientdet-d0 model with our scenario 4 pseudo-labels tfrecord.

```
bash /home/work/run_train.sh horovodrun -np 8 -H localhost:8
python ./efficientdet/main.py --mode train
                                --training_file_pattern *tfrecord
                                --model_name efficientdet-d0
                                --model_dir efficientdet-
d0/scenario_4/distillation
                                --save_dir s3://mybucket/myuser/model-
composition/output_models/scenario_4/distillation
                                --backbone_ckpt s3://mybucket/myuser/model-
composition/src/object_detection/efficientdet/noisy-efficientnet-b0
                                --train_batch_size 32
                                --num_examples_per_epoch 12268
                                --num_epochs 250
                                --strategy horovod
                                --cloud
                                --hparams 'num_classes=90'
```

Example of evaluating on cloud:

The following command shows how to evaluate the model trained in the above example.

```
bash /home/work/run_train.sh horovodrun -np 1 -H localhost:1
python ./efficientdet/main.py --mode eval_all
                                --validation_file_pattern val*tfrecord
                                --model_name efficientdet-d0
                                --model_dir efficientdet-
d0/scenario_4/eval_distillation
                                --save_dir s3://mybucket/myuser/model-
composition/output_models/scenario_4/distillation
                                --train_batch_size 32
                                --num_examples_per_epoch 12268
                                --num_epochs 250
                                --eval_samples 5000
```



```
--eval_batch_size 128
--strategy horovod
--cloud
--hparams 'num_classes=90'
```

Also specify the data storage location as the folder that contains the validation tfrecord:
S3://mybucket/datasets/MSCOCO2017-tfrecord/

Step 6 & 7: Finetune and evaluate

INPUTS:

Finetune and train are similar except that:

- `--backbone_ckpt` should be changed to `--ckpt`, which loads the best checkpoint in the composed model.

-- Note that we first copy the best checkpoint of the above composed training into a new folder, then specify the new folder as the `--ckpt`.

-- The best checkpoint should have 4 files: checkpoint files, data-00000-of-00001 file, index file, and meta file.

- `'var_exclude_expr=None, lr_warmup_epoch=0.0, finetune=False'` should be added to `--hparams` to make sure that heads of the composed model are loaded.
- `--num_examples_per_epoch` and `--num_epochs` should be re-calculated as below:

$$\text{num_examples_per_epoch} = \text{ceil}\left(\frac{\text{number of samples in finetune dataset}}{\text{number of gpus used for finetuning}}\right)$$

$$\text{num_epochs} = \text{ceil}\left(\frac{\left(\frac{(\text{number of finetune epochs} * \text{num_examples_per_epoch})}{\text{train_batch_size}} + \text{best ckpt step number}\right) * \text{train_batch_size}}{\text{num_examples_per_epoch}}\right)$$

For example, if the best checkpoint of the composed model is at step 10000, and we want to finetune it with COCO/train(size=118287) for 250 epochs, we will have:

$$\text{num_examples_per_epoch} = \text{ceil}(118287 / 8) = 14786$$

$$\text{num_epochs} = \text{ceil}(((250 * 14786 / 32) + 10000) * 32 / 14786) = 272$$

OUTPUTS:

All checkpoints and timeline json files

Example of finetuning on cloud:

We first copy the best checkpoint files into ["s3://mybucket/myuser/best_ckpt/scenario_4"](#), then use the following command to finetune our scenario 4 composed model with COCO data.

```
bash /home/work/run_train.sh horovodrun -np 8 -H localhost:8
python ./efficientdet/main.py --mode train
                                --training_file_pattern train*tfrecord
                                --model_name efficientdet-d0
                                --model_dir efficientdet-
d0/scenario_4/finetune_distillation
                                --save_dir s3://mybucket/myuser/model-
composition/output_models/scenario_4/finetune_with_coco
                                --ckpt
s3://mybucket/myuser/best_ckpt/scenario_4
                                --train_batch_size 32
                                --num_examples_per_epoch 14786
                                --num_epochs 272
                                --eval_samples 5000
                                --eval_batch_size 128
                                --strategy horovod
                                --cloud
                                --hparams 'num_classes=90, var_exclude_expr=None, lr_warmup_epoch=0.0, finetune=False'
```

Also specify the data storage location as the folder that contains the finetune train tfrecord: [S3://mybucket/datasets/MSCOCO2017-tfrecord/](#)

Note that if the finetune tfrecord is not already prepared before Step 6 & 7, please use a script in the `*efficientdet/dataset*` folder to create the tfrecord, or create a new script for a custom dataset first.

Example of evaluating the finetuned model on cloud:

The following command shows how to evaluate the finetuned model.

```
bash /home/work/run_train.sh horovodrun -np 1 -H localhost:1
```

```
python ./efficientdet/main.py --mode eval_all
                                --validation_file_pattern val*tfrecord
                                --model_name efficientdet-d0
                                --model_dir efficientdet-
d0/scenario_4/eval_finetune_distillation
                                --save_dir s3://mybucket/myuser/model-
composition/output_models/scenario_4/finetune_with_coco
                                --train_batch_size 32
                                --num_examples_per_epoch 14786
                                --num_epochs 272
                                --strategy horovod
                                --cloud
                                --hparams 'num_classes=90, var_exclude_expr=None, lr_warmup_epoch=0.0, finetune=False'
```

Also specify the data storage location as the folder that contains the finetune validation tfrecord: S3://mybucket/datasets/MSCOCO2017-tfrecord/

Model Composition for Classification

Pre-requisites

- Tensorflow 2.1
- CUDA 10.1

Setting-up locally

1. Create conda environment and install dependencies

```
conda env create -f modelcomp-tf21.yaml
```

Setting-up on cloud

1. Create custom cloud image using *Dockerfile*.
Required files for creating custom image:
 - [tensorflow_gpu-2.1.0-cp36-cp36m-manylinux2010_x86_64.whl](#)
2. If **imagenet** weights are needed, cloud can't automatically download the weights from internet, so they have to be downloaded beforehand. They can be downloaded following URLs from this [Weights Collection](#). Then adjust the WEIGHTS_STUDENT parameter in *params.py*. Otherwise set to None for training from scratch.
3. Change the **CLOUD** parameter to 'True' in *params.py*

An existing custom image on cloud is: `tf21_trt6, 1.0.0`

How to Run

Script Arguments:

model_path_00: Path to the first model to combine.

model_path_01: Path to the second model to combine.

model_path_02: Path to the third model to combine. Set to None in scenario 3.

target_model: Target model architecture (i.e. resnet18, resnet50, etc.).

data_url: (Optional) Path to original dataset (used for fine-tuning).

eval_url: (Optional) Path to validation/test dataset (usually, validation set of dataset used to train teacher/input model).

data_to_label: Path to dataset used for generating pseudo-labels.

output_name: Name of input model's probabilities output layer (i.e. output/Softmax:0).

labels_file: Path to text file containing mappings of label index to label name.

image_size: (Optional. Default=384) Size of model input. Only squared images allowed in this version.

More training, combination and cloud related parameters are available at **params.py** file

Locally:

```
python combine_tf2.py --model_path_00 /path/to/first/sub-model.h5
                    --model_path_01 /path/to/second/sub-model.h5
                    --model_path_02 /path/to/third/sub-model.h5
                    --target_model /target/model/type (i.e. resnet18,
resnet50 etc.)
                    --data_url /dataset/path/train
                    --eval_url /dataset/path/val
                    --data_to_label /dataset/path
                    --labels_file /dataset/path/labels.txt
                    --image_size /size/of/input/image
                    --output_name output/Softmax:0
```

Note:

- data_url and eval_url are optional. If not given, fine-tuning and validation won't be performed, but model will still be distilled and combined.
- image_size is optional. If not given, is set to 384 by default.

In cloud (custom image):

```
bash /home/work/run_train.sh python /home/work/user-job-
dir/classification/combine_tf2.py
                    --model_path_00 s3://<BUCKET>/path/to/first/model.h5
                    --model_path_01 s3://<BUCKET>/path/to/second/model.h5
                    --model_path_02 s3://<BUCKET>/path/to/third/model.h5
                    --target_model /target/model/type (i.e. resnet18, resnet50 etc.)
                    --data_url s3://<BUCKET>/dataset/path/train
                    --eval_url s3://<BUCKET>/dataset/path/val
                    --data_to_label s3://<BUCKET>/dataset/path
                    --labels_file s3://<BUCKET>/dataset/path/labels.txt
```

```
--image_size /size/of/input/image  
--output_name output/Softmax:0
```

Testing

Combined models can be tested using the test script.

```
python test.py --model_path /path/to/combined/model.h5  
--data_path /dataset/path/test  
--output_layers output/Softmax:0  
--labels_file /dataset/path/labels.txt
```

Supported models

Input Models

Input model format can be either .PB graphs or Keras .H5 models

Model output layer name has to be specified when running the script (i.e. 'Softmax:0')

Target Models

Any model supported by the image-classifiers package can be used as target/student model.

Available models: *resnet18, resnet34, resnet50, resnet101, resnet152, seresnet18, seresnet34, seresnet50, seresnet101, seresnet152, seresnext50, seresnext101, senet154, resnet50v2, resnet101v2, resnet152v2, resnext50, resnext101, vgg16, vgg19, densenet121, densenet169, densenet201, inceptionresnetv2, inceptionv3, xception, nasnetlarge, nasnetmobile, mobilenet, mobilenetv2*

For mobilenet architectures, network depth can be selected by changing the **DEPTH_FACTOR** in params.py

Outputs

- Pseudo-labeling process outputs a csv file named *combined_labels.csv* into the labels output folder containing file names and predicted classes;
If using stacking as the combination method, there will be 3 other csv files named *labeled_00.csv, labeled_01.csv, labeled_02.csv* containing file names and predicted classes of each sub-model.

- Training process outputs tensorboard logs, as well as models in h5 and pb format
- The final combined model will be output to the final training step folder. If there's finetune available, it will be *finetune*, otherwise it will be *train*

Experiments

Brief Intro

* OID used in all experiments is a subset of the original OID

Scenario 1

- Train 3 resnet18 models, each on 1/3 of Caltech256 training set
- Generate pseudo-labels using these models on entire Caltech256 training set or entire OID* (train+validation+test)
- Combine pseudo-labels using various ensembling methods
- Train a new resnet18 model on the combined dataset
- Finetune the new model with subset of Caltech256
- Evaluate the combined model on Caltech256 validation set

Scenario 2

- Train 3 models: resnet18, resnet152 and densenet121 on entire Caltech256 training set
- Generate pseudo-labels using these models on entire Caltech256 training set or on entire OID* (train+validation+test)
- Combine pseudo-labels using various ensembling methods
- Train a new model on the combined dataset (using 3 backbones: R-18, R-50, R-152)
- Finetune the new model with subset of Caltech256
- Evaluate the combined model on Caltech256 validation set

Scenario 3

- Train 2 resnet18 models, one on 1/2 of Caltech256 training set, the other on 1/2 of the union of OID* train and test set
- Generate pseudo-labels using these models on the union of the other 1/2 of Caltech256 and OID*
- Combine pseudo-labels using various ensembling methods
- Train a new resnet18 model on the combined dataset

- Finetune the new model with subset of a. the first 1/2 of Caltech256(or OID*) b. the second 1/2 of Caltech256(or OID*) c. full Caltech256(or OID*)
- Evaluate on either Caltech256 validation set or on OID* validation set

Note:

The first 1/2 of Caltech256/OID* are what the 2 baseline models are trained on