AssembleNet and AssembleNet++

This repository is the official implementations of the following papers.

The original implementations could be found in [here]

arXiv Paper arXiv.2008.03800 [AssembleNet: Searching for Multi-Stream Neural Connectivity in Video Architectures]

arXiv Paper arXiv.2008.08072 [AssembleNet++: Assembling Modality Representations via Attention Connections]

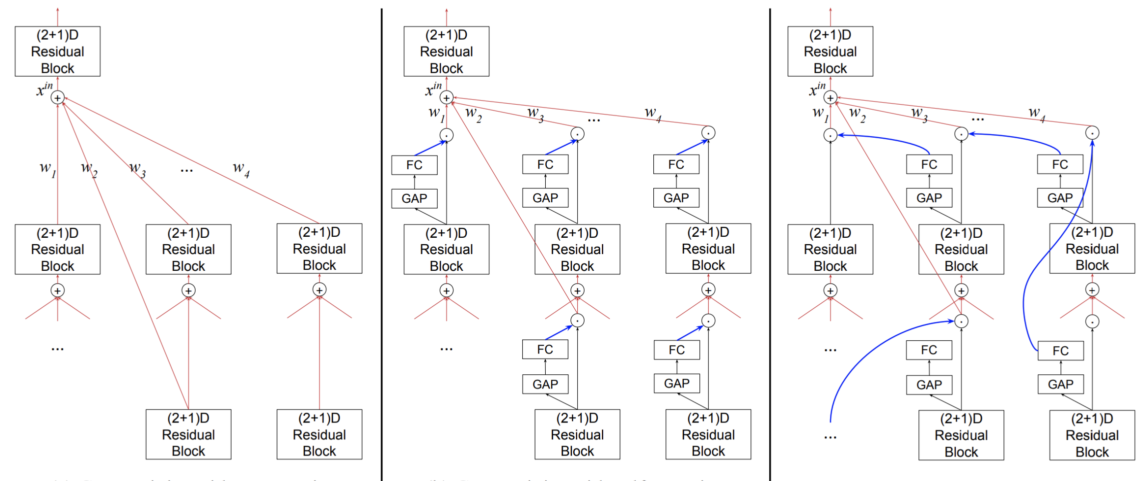
DISCLAIMER: AssembleNet++ implementation is still under development. No support will be provided during the development phase.

Description

AssembleNet vs. AssembleNet++

AssembleNet and AssembleNet++ both focus on neural connectivity search for multi-stream video CNN architectures. They learn weights for the connections between multiple convolutional blocks (composed of (2+1)D or 3D residual modules) organized sequentially or in parallel, thereby optimizing the neural architecture for the data/task.

AssembleNet++ adds *peer-attention* to the basic AssembleNet, which allows each conv. block connection to be conditioned differently based on another block. It is a form of channel-wise attention, which we found to be beneficial.



(a) Connectivity without attention (b) Connectivity with self-attention (c) Connectivity with peer-attention

The code is provided in [assemblenet.py] and [assemblenet_plus.py]. Notice that the provided code uses (2+1)D residual modules as the building blocks of AssembleNet/++, but you can use your own module while still benefitting from the connectivity search of AssembleNet/++.

Neural Architecture Search

As you will find from the AssembleNet paper, the models we provide in config files are the result of architecture search/learning.

The architecture search in AssembleNet (and AssembleNet++) has two components: (i) convolutional block configuration search using an evolutionary algorithm, and (ii) one-shot differentiable connection search. We did not include the code for the first part (i.e., evolution), as it relies on another infrastructure and more computation. The 2nd part (i.e., differentiable search) is included in the code however, which will allow you to use to code to search for the best connectivity for your own models.

That is, as also described in the AssembleNet++ paper, once the convolutional blocks are decided based on the search or manually, you can use the provide code to obtain the best block connections and learn attention connectivity in a one-shot differentiable way. You just need to train the network (with `FLAGS.model_edge_weights` as `[]`) and the connectivity search will be done simultaneously.

AssembleNet and AssembleNet++ Structure Format

The format we use to specify AssembleNet/++ architectures is as follows: It is a `list` corresponding to a graph representation of the network, where a node is a convolutional block and an edge specifies a connection from one block to another. Each node itself (in the structure list) is a `list` with the following format: `[block_level, [list_of_input_blocks], number_filter, temporal_dilation, spatial_stride]`. `[list_of_input_blocks]` should be the list of node indexes whose values are less than the index of the node itself. The 'stems' of the network directly taking raw inputs follow a different node format: `[stem_type, temporal_dilation]`. The stem_type is -1 for RGB stem and is -2 for optical flow stem. The stem_type -3 is reserved for the object segmentation input.

In AssembleNet++lite, instead of passing a single `int` for `number_filter`, we pass a list/tuple of three `int`s. They specify the number of channels to be used for each layer in the inverted bottleneck modules.

Optical Flow and Data Loading

Instead of loading optical flows as inputs from data pipeline, we are applying the Representation Flow to RGB frames so that we can compute the flow within TPU/GPU on fly. It's essentially optical flow since it is computed directly from RGBs. The benefit is that we don't need an external optical flow extraction and data loading. You only need to feed RGB, and the flow will be computed internally.

History

2021/10/02 : AssembleNet, AssembleNet++ implementation with UCF101 dataset provided

Authors

- SunJong Park (@GitHub ryan0507)
- HyeYoon Lee (@GitHub hylee817)

Table of Contents

Requirements

TensorFlow 2.5.0 Python 3.8

Training and Evaluation

Example of training AssembleNet with UCF101 TF Datasets.

```
python -m official.vision.beta.projects.assemblenet.trian \
--mode=train_and_eval --experiment=assemblenet_ucf101 \
--model_dir='YOUR_GS_BUCKET_TO_SAVE_MODEL' \
--config_file=./official/vision/beta/projects/assemblenet/\
--ucf101_assemblenet_tpu.yaml \
--tpu=TPU_NAME
```

Example of training AssembleNet++ with UCF101 TF Datasets.

```
python -m official.vision.beta.projects.assemblenet.trian \
--mode=train_and_eval --experiment=assemblenetplus_ucf101 \
--model_dir='YOUR_GS_BUCKET_TO_SAVE_MODEL' \
--config_file=./official/vision/beta/projects/assemblenet/\
--ucf101_assemblenet_plus_tpu.yaml \
--tpu=TPU_NAME
```

Currently, we provide experiments with kinetics400, kinetics500, kinetics600, UCF101 datasets. If you want to add a new experiment you should modify exp_factory for configuration.