MSCG Net

Henry Lao, Manish Kakarla, Pradeep Gummidipundi, Atiya Kailany

Dec 04, 2021

OVERVIEW

sphinx-quickstart on Wed Nov 24 07:25:06 2021. You can adapt this file completely to your liking, but it should at least contain the root *toctree* directive.

This project was designed to adapt the work of Liu et al. and convert it to a mobile device which would allow for in-the-field processing of images and much faster response time and lower network requirements versus a computing cluster that would typically be utilized for these sorts of tasks. While working on this adaptation, we utilized two separate methods to ensure flexibility of classification: a local method powered entirely by the Android device for those phones with the computing capacity to spare, and a REST-based method designed to take advantage of existing networks and send the image back to a computer for offsite processing, storage, and evaluation. The following paper describes implementation, downloading and running instructions, screenshots, and finally comments/critiques.

CHAPTER

ONE

DEVELOPMENT

1.1 Quickstart

1.2 Installation

1.3 Configuration

1.4 Development

1.5 Changelog

1.6 Overview

1.7 Demo

1.8 Preprocessing

1.9 Models

1.10 Deployment

1.11 Overview

1.12 Installation

1.13 Usage

1.14 Overview

1.15 Demo

1.16 Preprocessing

utils.data.augmentation.**get_random_pos**(*img*, *window_shape*)
Extract of 2D random patch of shape window_shape in the image

utils.data.augmentation.**pad_tensor**(*image_tensor: torch.Tensor*, *pad_size: int = 32*)
Pads input tensor to make it's height and width dividable by @pad_size

Parameters

- **image_tensor** – Input tensor of shape NxCxHxW
- **pad_size** – Pad size

Returns Tuple of output tensor and pad params. Second argument can be used to reverse pad operation of metrics output

utils.data.augmentation.**rm_pad_tensor**(*image_tensor*, *pad*)
Remove padding from a tensor

Parameters

- **image_tensor** –
- **pad** –

Returns

1.17 Models

class core.net.**RX50GCN3Head4Channel**(*out_channels=7*, *pretrained=True*, *nodes=(32, 32)*, *dropout=0*, *enhance_diag=True*, *aux_pred=True*)

__init__(*out_channels=7*, *pretrained=True*, *nodes=(32, 32)*, *dropout=0*, *enhance_diag=True*, *aux_pred=True*)

Parameters

- **out_channels** –
- **pretrained** –
- **nodes** –
- **dropout** –
- **enhance_diag** –
- **aux_pred** –

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class core.net.**RX101GCN3Head4Channel**(*out_channels=7*, *pretrained=True*, *nodes=(32, 32)*, *dropout=0*, *enhance_diag=True*, *aux_pred=True*)

__init__(*out_channels=7*, *pretrained=True*, *nodes=(32, 32)*, *dropout=0*, *enhance_diag=True*, *aux_pred=True*)

Parameters

- **out_channels** –
- **pretrained** –
- **nodes** –
- **dropout** –
- **enhance_diag** –
- **aux_pred** –

apply(*fn*)

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

forward(*x*)

Parameters **x** –

Returns

class core.net.**SCGBlock**(*in_ch*, *hidden_ch=6*, *node_size=(32, 32)*, *add_diag=True*, *dropout=0.2*)

__init__(*in_ch*, *hidden_ch=6*, *node_size=(32, 32)*, *add_diag=True*, *dropout=0.2*)

Self-Constructing Graph module facilitating construction of undirected graphs

Module for creating undirected graphs and capturing relations across images from feature maps (weight adjacency matrices) by learning the **mean matrix** and a **standard deviation matrix** of a Gaussian using 2 single-layer **CNNs**. Parameter free adaptive average pooling is used to reduce the spatial dimensions of the input. Usage of diagonal regularization is applied to stabilize training and to preserve local information. The output of the of module is a symmetric adjacency matrix and an adaptive residual prediction (used to refine the final prediction after information propogation along the graph)

Feature Map

$$X \in \mathbb{R}^{h \times w \times d}$$

Graph of Converted Feature Map

$$G = (\hat{A}, X') \mid X' \in \mathbb{R}^{n \times d}, n = h' \times w' \mid (h' \times w')(h \times w)$$

Standard deviation of the output

$$\log(\sigma)$$

of the module follows the convention of variational autoencoders to ensure stability during training

Mean Matrix

$$\mu \in \mathbb{R}^{n \times c}$$

Standard Deviation Matrix

$$\sigma \in \mathbb{R}^{n \times c}$$

Latent Embedding

$$Z \leftarrow \mu + \sigma \cdot \epsilon \mid \epsilon \in \mathbb{R}^{\mathbb{N}' \times \mathbb{C}}$$

Auxiliary Noise initialized from a standard normal distribution

$$\epsilon \in \mathbb{R}^{\mathbb{N}' \times \mathbb{C}} \mid \epsilon \sim N(0, 1)$$

From the learned latent embeddings, we have activation .. math:: A' computed as

$$A' = \text{ReLU}(ZZ^T)$$

such that activations

$$A'_{ij} > 0$$

denote the presence of an edge between the nodes

$$i, j$$

Usage of diagonal regularization for stabilizing training and preservie local information

forward(*x*)

classmethod laplacian_matrix(*A*, *self_loop=False*)
Computes normalized Laplacian matrix: A (B, N, N)

class core.net.**GCNLayer**(*in_features*, *out_features*, *bnorm=True*, *activation=ReLU()*, *dropout=None*)

__init__(*in_features*, *out_features*, *bnorm=True*, *activation=ReLU()*, *dropout=None*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*data*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class core.net.**BatchNormGCN**(*num_features*)
Batch normalization over GCN features

__init__(*num_features*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

1.18 Utilities

utils.__init__.**check_mkdir**(*dir_name: str*) → None
Utility function that creates a directory if the path does not exist

Parameters **dir_name** – str

Returns

1.18.1 Tracing

Utility functions for model debugging, setup and loading

Checkpoint

GPU

utils.gpu.**get_available_gpus**(*memory_threshold: float = 0.0*, *metric: str = 'mb'*) → List
Get all the available GPUs using less memory than a specified threshold

Parameters

- **memory_threshold** – maximum memory usage threshold to reject
- **metric** – GB or MB

Returns List

utils.gpu.**get_memory_map**() → dict
Get the current gpu usage.

Returns **usage** – Keys are device ids as integers. Values are memory usage as integers in MB.

Return type dict

utils.gpu.**get_stats**() → pandas.core.frame.DataFrame
Get statistics of all GPUs in a DataFrame

Returns

Logger

utils.logger.**setup_logger**(*log_directory: str*, *model_name: str*) → None
Function for setting up the logger for debugging purposes

Parameters

- **log_directory** –
- **model_name** –

Returns

utils.logger.**tracer**(*func*)
Decorator to print function call details :param func: :return:

Metrics

Loss

class utils.metrics.loss.**ACWLoss**(*ini_weight=0*, *ini_iteration=0*, *eps=1e-05*, *ignore_index=255*)

__init__(*ini_weight=0*, *ini_iteration=0*, *eps=1e-05*, *ignore_index=255*)
Adaptive Class Weighting Loss is the loss function class for handling the highly imbalanced distribution of images Multi-class adaptive class loss function

Adaptive Class Weighting Loss

$$L_{acw} = \frac{1}{|Y|} \sum_{i \in Y} \sum_{j \in C} \tilde{w}_{ij} \times p_{ij} - log(\text{ MEAN}\{d_j | j \in C\})$$

Dice coefficient

$$d_j = \frac{2\sum_{i \in Y} y_{ij}\tilde{y}_{ij}}{\sum_{ij} y_{ij} + \sum_{i \in Y} \tilde{y}_{ij}}$$

Parameters

- **ini_weight** –
- **ini_iteration** –
- **eps** –

:param ignore_index:z

adaptive_class_weight(*pred*, *one_hot_label*, *mask=None*)

Adaptive Class Weighting (ACW) computed based on the iterative batch-wise class derived from the median frequency to balance weights.

ACW

$$\tilde{w}_{ij} = \frac{w_j^t}{\sum_{j \in C}(w_j^t)} \times (1 + y_{ij} + \tilde{y}_{ij})$$

Iterative Median Frequency Class Weights

$$w_j^t = \frac{\text{MEDIAN}(\{f_j^t | j \in C\})}{f_j^t + \epsilon} \mid \epsilon = 10^{-5}$$

Pixel Frequency

$$f_j^t = \frac{\hat{f}_j^t + (t-1) \times f_j^{t-1}}{t} \mid t \in \{1, 2, ..., \infty\}$$

Parameters

- **pred** –
- **one_hot_label** –
- **mask** –

Returns

forward(*prediction*, *target*)

pred : shape (N, C, H, W) target : shape (N, H, W) ground truth return: loss_acw

pnc(*err*)

Apply positive-negative class balanced function (PNC)

PNC

$$p = e - \log\left(\frac{1-e}{1+e}\right) \mid e = (y - \tilde{y})^2$$

Parameters **err** –

Returns

Optimizer

class utils.metrics.optimizer.**Lookahead**(*base_optimizer*, *alpha=0.5*, *k=6*)

load_state_dict(*state_dict*)
Loads the optimizer state.

Parameters **state_dict** (*dict*) – optimizer state. Should be an object returned from a call to *state_dict()*.

state_dict()
Returns the state of the optimizer as a dict.

It contains two entries:

- **state - a dict holding current optimization state. Its content** differs between optimizer classes.
- **param_groups - a list containing all parameter groups where each** parameter group is a dict

step(*closure=None*)
Performs a single optimization step (parameter update).

Parameters **closure** (*callable*) – A closure that reevaluates the model and returns the loss. Optional for most optimizers.

Note: Unless otherwise specified, this function should not modify the .grad field of the parameters.

Learning Rate

utils.metrics.lr.**adjust_initial_rate**(*optimizer*, *i_iter*, *opt*, *model='cos'*)
Function for adjusting scheduling learning rate in accordance to a specified model with the provided optimizer

Parameters

- **optimizer** –
- **i_iter** –
- **opt** –
- **model** – “cos” denotes cosine annealing to reduce lr over epochs

Returns

utils.metrics.lr.**adjust_learning_rate**(*optimizer*, *i_iter*, *opt*)

Parameters

- **optimizer** –
- **i_iter** –
- **opt** –

Returns

utils.metrics.lr.**init_params_lr**(*net*, *opt*)

Parameters

- **net** –
- **opt** –

Returns

utils.metrics.lr.**lr_cos**(*base_lr*, *iteration*, *max_iterations*)

Parameters

- **base_lr** –
- **iteration** –
- **max_iterations** –

Returns

utils.metrics.lr.**lr_poly**(*base_lr*, *iteration*, *max_iterations*, *power*)

Parameters

- **base_lr** –
- **iteration** –
- **max_iterations** –
- **power** –

Returns

Validate

utils.metrics.validate.**evaluate**(*predictions*, *gts*, *num_classes*)
Function for evaluating the collection of predictions given the set of ground-truths

Parameters

- **predictions** –
- **gts** –
- **num_classes** –

Returns

utils.metrics.validate.**multiprocess_evaluate**(*predictions*, *gts*, *num_classes*)
Function for evaluating the collection of predictions given the set of ground-truths

Parameters

- **predictions** –
- **gts** –
- **num_classes** –

Returns

1.18.2 Export

Android

utils.export.android.**convert_to_mobile**(*model: str*, *source_path: str*, *output_path: str*, *num_classes: int*) → torch.nn.modules.module.Module

Main function for converting MSCG core to PyTorch Mobile

NOTE Usage of PyTorch Mobile to convert the MSCG-Nets requires usage of a matching Android PyTorch Mobile Version 1.10

Parameters

- **num_classes** –
- **model** –
- **source_path** –
- **output_path** –

Returns

Visualizations

Configuration

1.19 Agriculture Vision 2021

1.19.1 Results Summary

NOTE all our single model's scores are computed with just single-scale (512x512) and single feed-forward inference without TTA. TTA denotes test time augmentation (e.g. flip and mirror). Ensemble_TTA (checkpoint1,2) denotes two core.net.(checkpoint1, and checkpoint2) ensemble with TTA, and (checkpoint1, 2, 3) denotes three core.net.ensemble.

Models	mIoU (%)	Back-ground	Cloud shadow	Double plant	Planter skip	Standing water	Wa-ter-way	Weed cluster
MSCG-Net-50 (ckpt1)	54.7	78.0	50.7	46.6	34.3	68.8	51.3	53.0
MSCG-Net-101 (ckpt2)	***55.0***	***79.8***	***44.8***	***55.0***	***30.5***	***65.4***	***59.2***	***50.6***
MSCG-Net-101_k31 (ckpt3)	54.1	79.6	46.2	54.6	9.1	74.3	62.4	52.1
Ensemble_TTA (ckpt1,2)	59.9	80.1	50.3	57.6	52.0	69.6	56.0	53.8
Ensemble_TTA (ckpt1,2,3)	60.8	80.5	**51.0**	58.6	49.8	**72.0**	59.8	**53.8**
Ensemble_TTA (new_5model)	**62.2**	**80.6**	48.7	**62.4**	**58.7**	71.3	**60.1**	53.4

1.19.2 Model Size

NOTE all backbones used pretrained weights on **ImageNet** that can be imported and downloaded from the link. And MSCG-Net-101_k31 has exactly the same architecture wit MSCG-Net-101, while it is trained with extra 1/3 validation set (4,431) instead of just using the official training images (12,901).

Models	Backbones	Parameters	GFLOPs	Inference time (CPU/GPU)
MSCG-Net-50	Se_ResNext50_32x4d	9.59	18.21	522 / 26 ms
MSCG-Net-101	Se_ResNext101_32x4d	30.99	37.86	752 / 45 ms
MSCG-Net-101_k31	Se_ResNext101_32x4d	30.99	37.86	752 / 45 ms

1.20 Agriculture Vision 2020

1.20.1 Results Summary

NOTE all our single model's scores are computed with just single-scale (512x512) and single feed-forward inference without TTA. TTA denotes test time augmentation (e.g. flip and mirror). Ensemble_TTA (checkpoint1,2) denotes two core.net.(checkpoint1, and checkpoint2) ensemble with TTA, and (checkpoint1, 2, 3) denotes three core.net.ensemble.

Models	mIoU (%)	Back-ground	Cloud shadow	Double plant	Planter skip	Standing water	Wa-ter-way	Weed cluster
MSCG-Net-50 (ckpt1)	54.7	78.0	50.7	46.6	34.3	68.8	51.3	53.0
MSCG-Net-101 (ckpt2)	***55.0***	***79.8***	***44.8***	***55.0***	***30.5***	***65.4***	***59.2***	***50.6***
MSCG-Net-101_k31 (ckpt3)	54.1	79.6	46.2	54.6	9.1	74.3	62.4	52.1
Ensemble_TTA (ckpt1,2)	59.9	80.1	50.3	57.6	52.0	69.6	56.0	53.8
Ensemble_TTA (ckpt1,2,3)	60.8	80.5	**51.0**	58.6	49.8	**72.0**	59.8	**53.8**
Ensemble_TTA (new_5model)	**62.2**	**80.6**	48.7	**62.4**	**58.7**	71.3	**60.1**	53.4

1.20.2 Model Size

NOTE all backbones used pretrained weights on **ImageNet** that can be imported and downloaded from the link. And MSCG-Net-101_k31 has exactly the same architecture wit MSCG-Net-101, while it is trained with extra 1/3 validation set (4,431) instead of just using the official training images (12,901).

Models	Backbones	Parameters	GFLOPs	Inference time (CPU/GPU)
MSCG-Net-50	Se_ResNext50_32x4d	9.59	18.21	522 / 26 ms
MSCG-Net-101	Se_ResNext101_32x4d	30.99	37.86	752 / 45 ms
MSCG-Net-101_k31	Se_ResNext101_32x4d	30.99	37.86	752 / 45 ms

PYTHON MODULE INDEX

u

INDEX

Symbols

A

B

C

E

F

G

I

L

M

P

R

S

T

U