cords

Release v1.0

Krishnateja Killamsetty, Dheeraj Bhat, Rishabh Iyer

Feb 23, 2022

CONTENTS:

CHAPTER

ONE

CORDS::CORESETS AND DATA SUBSET SELECTION

CORDS:: COResets and Data Subset slection is an efficient and scalable library for making machine learning time, energy, cost, and compute efficient built on top of pytorch.

Background: Deep Learning systems are extremely compute-intensive today, with significant turnaround times, energy inefficiencies leading to huge resource costs. Another aspect of deep learning systems is that they have a surprisingly large carbon footprint with lasting environmental impacts(see https://arxiv.org/pdf/1907.10597.pdf and https://arxiv.org/abs/1906.02243 for more details on quantifications of these impacts). Furthermore, to achieve the state of the art performances using deep learning models, we need to perform hyper-parameter tuning (which requires training a deep learning model multiple times).

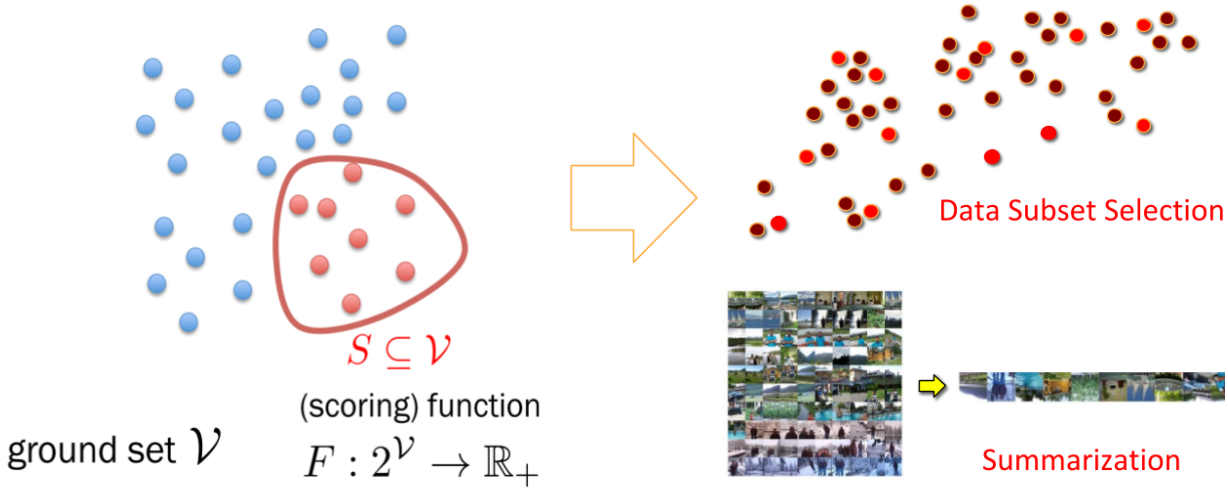
CORDS is an effort to make deep learning more energy, cost, resource and time efficient while not sacrificing accuracy.

The following are the goals CORDS tries to achieve:

- Data Efficiency
- Reducing End to End Training Time
- Reducing Energy requirements
- Reducing Hyper-parameter tuning turnaround time
- Reducing Resource (GPU) requirements and Costs

CORDS' key idea is to use a subset of training data for model training. In this effort, CORDS incorporate various state-of-the-art data selection strategies to select the proper representative data subset from massive datasets.

The selected subsets can be used for summarizing the dataset like finding the key parts of a video.



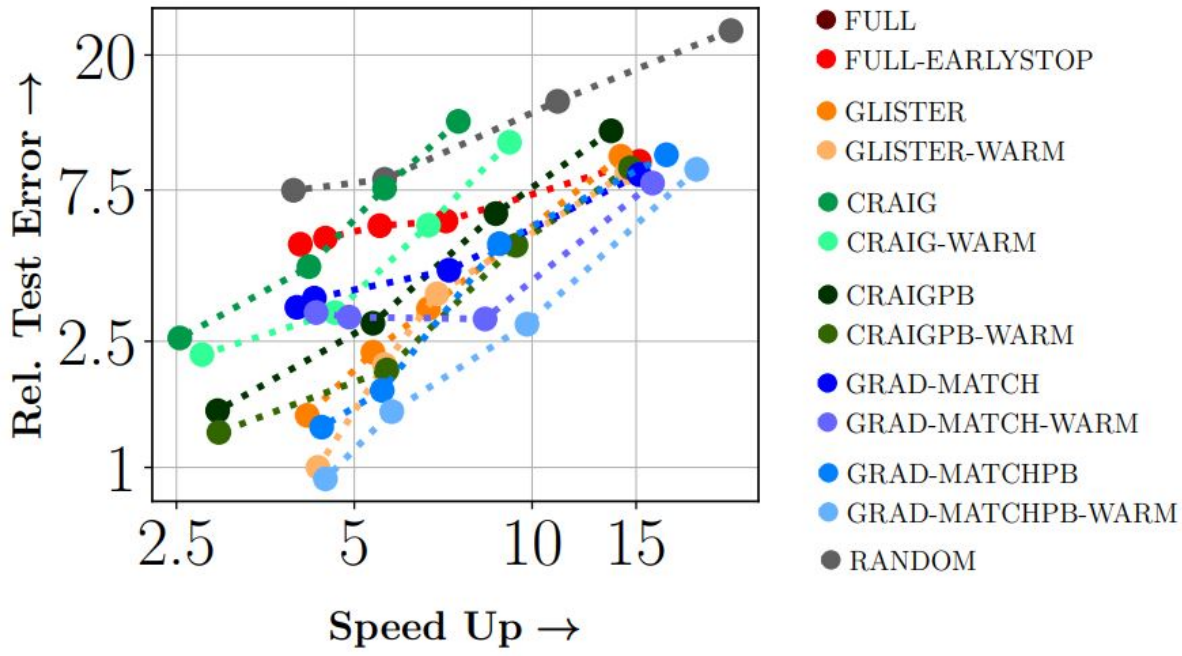
These subsets can also be used for training machine learning models for faster and efficient training. It might seem counterintuitive at first to train a model using only a fraction of your data. Unfortunately, the compute required to train models on huge data sets might not be available to everyone.

Attention: Training a Yolo V5X Model may take 8 days on a single V-100 GPU.

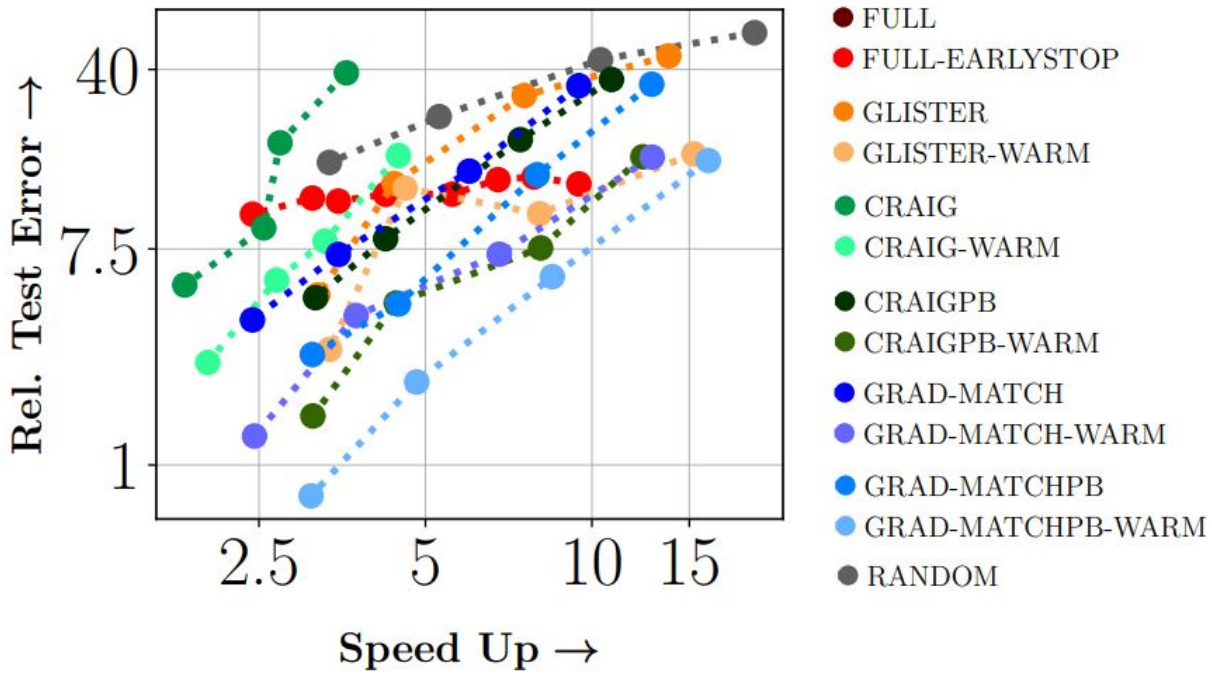
Instead of relying on random subsampling, one could instead select a subset using various data selection strategies. The CORDS repository contains some of the state of the art data subset selection strategies that achieves close to full training accuracy even when trained on a meager 10% subset of data while achieving significant speed ups.

Results: Results show 3x to 7x improvements in energy and runtime with around 1 - 2% drop in accuracy. We expect to push the Pareto-optimal frontier even more over time.

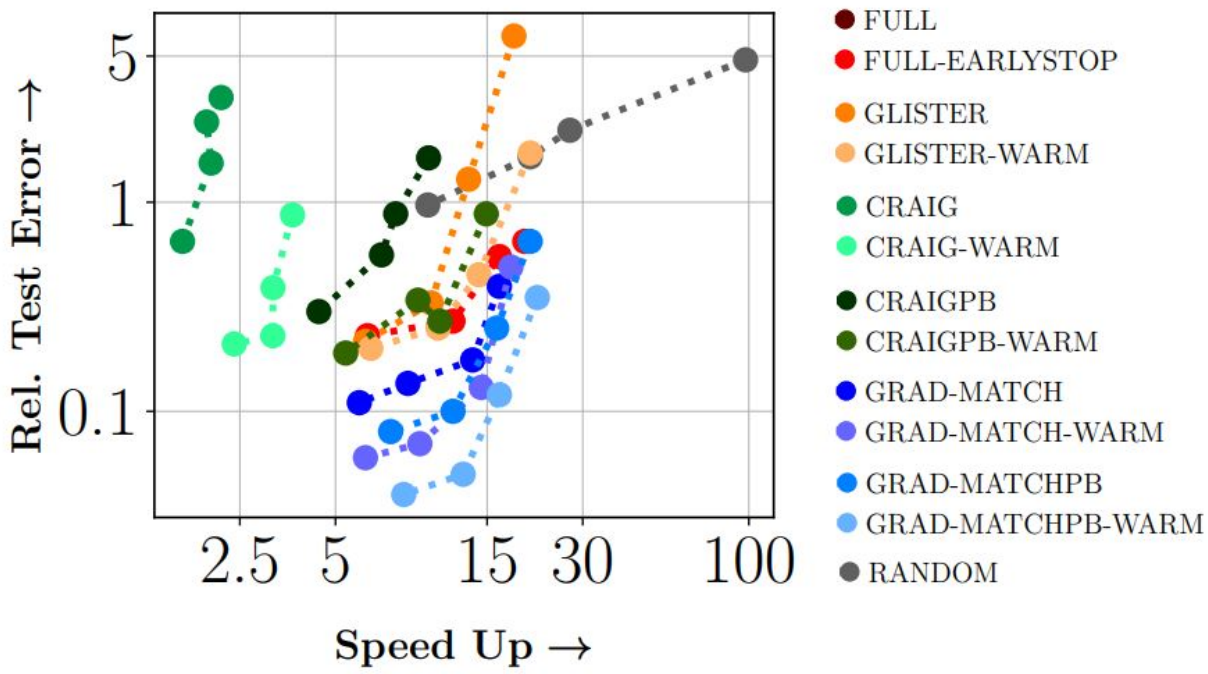
Cifar10


Rel. Test Error →
20
7.5
2.5
1
2.5
5
10
15
Speed Up →
FULL
FULL-EARLYSTOP
GLISTER
GLISTER-WARM
CRAIG
CRAIG-WARM
CRAIGPB
CRAIGPB-WARM
GRAD-MATCH
GRAD-MATCH-WARM
GRAD-MATCHPB
GRAD-MATCHPB-WARM
RANDOM

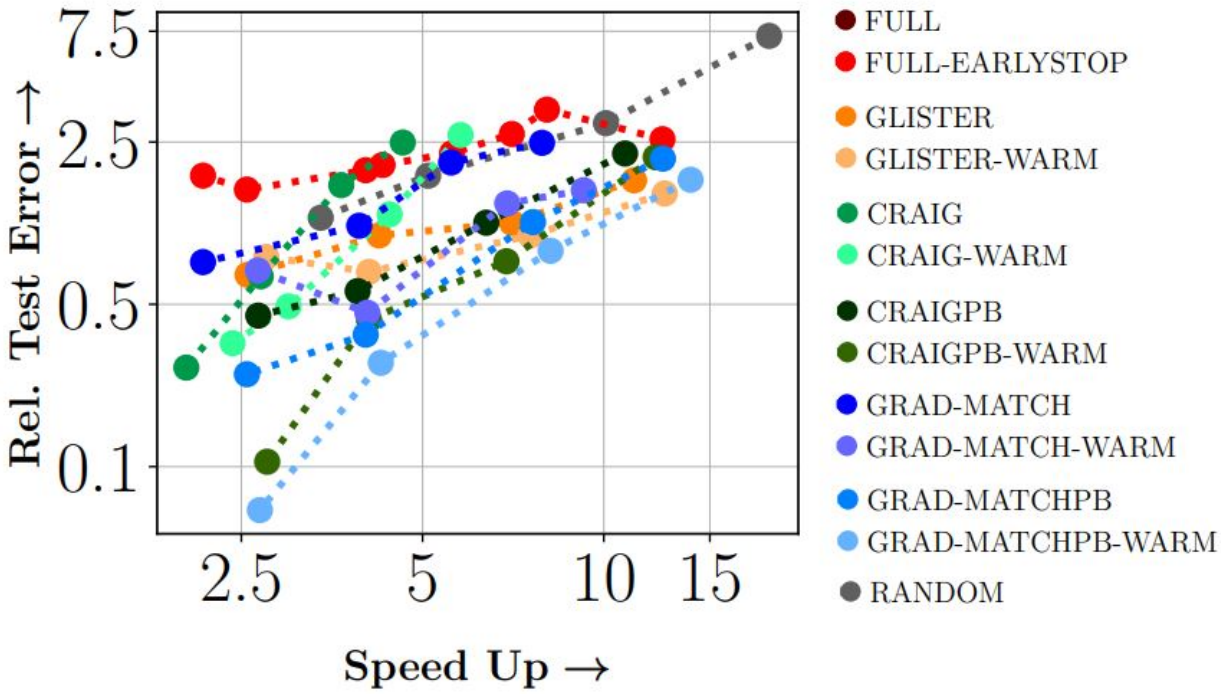
Cifar100


Rel. Test Error →
40
7.5
1
2.5
5
10
15
Speed Up →
FULL
FULL-EARLYSTOP
GLISTER
GLISTER-WARM
CRAIG
CRAIG-WARM
CRAIGPB
CRAIGPB-WARM
GRAD-MATCH
GRAD-MATCH-WARM
GRAD-MATCHPB
GRAD-MATCHPB-WARM
RANDOM

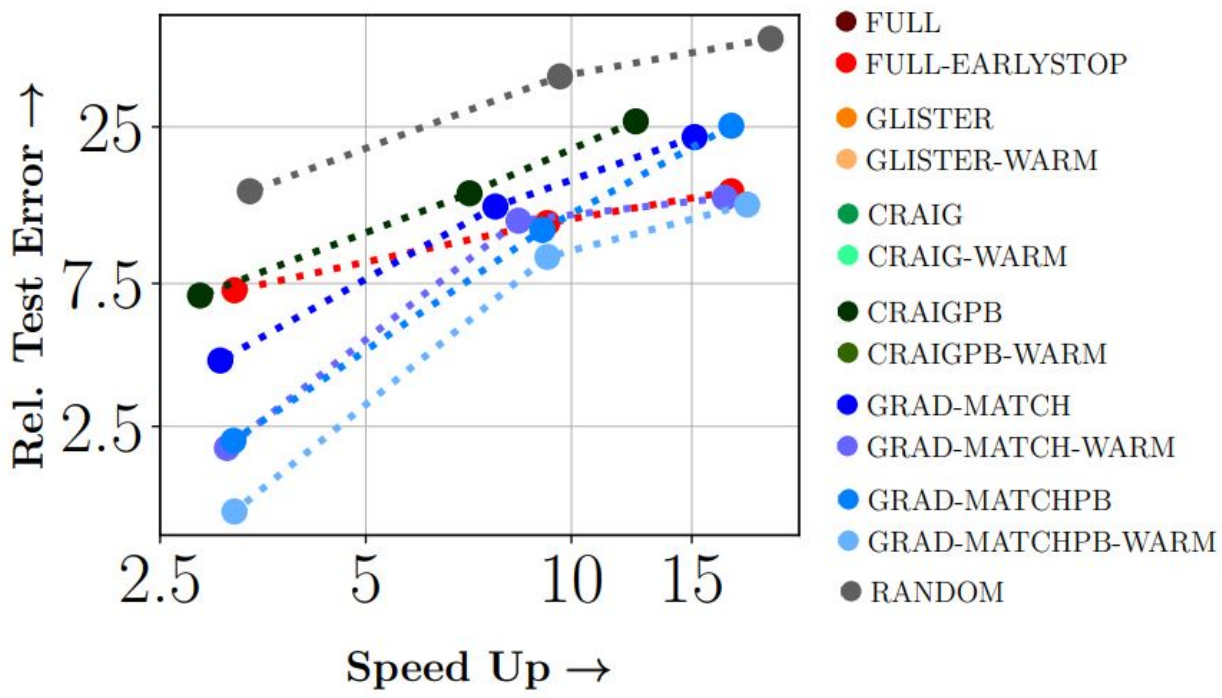
MNIST


Rel. Test Error →
5
1
0.1
2.5
5
15
30
100
Speed Up →
FULL
FULL-EARLYSTOP
GLISTER
GLISTER-WARM
CRAIG
CRAIG-WARM
CRAIGPB
CRAIGPB-WARM
GRAD-MATCH
GRAD-MATCH-WARM
GRAD-MATCHPB
GRAD-MATCHPB-WARM
RANDOM

SVHN


7.5
2.5
0.5
0.1
Rel. Test Error →
2.5
5
10
15
Speed Up →
FULL
FULL-EARLYSTOP
GLISTER
GLISTER-WARM
CRAIG
CRAIG-WARM
CRAIGPB
CRAIGPB-WARM
GRAD-MATCH
GRAD-MATCH-WARM
GRAD-MATCHPB
GRAD-MATCHPB-WARM
RANDOM

ImageNet



Note: From the above figure, we can see that on CIFAR-10 dataset using RESNET18, GLISTER a data subset selection strategy achieves a 6x speedup at 10%, 2.5x speedup at 30%, and 1.5x speedup at 50%, while losing 3%, 1.2% and 0.2% in terms of accuracy compared to full training.

1.1 CORDS

1.1.1 Data Selection Strategies

We are working on bringing the data subset selection strategies for various machine learning frameworks like:

- Supervised learning (SL)
- Semi-supervised learning (SSL)

In our current version, we deployed data subset selection strategies in supervised learning and semi-supervised learning settings.

Supervised Learning Data Selection Strategies

In this section, we consider different data selection strategies geared towards efficient and robust learning in standard supervised learning setting.

Data Selection Strategy (Base Class)

class cords.selectionstrategies.SL.dataselectionstrategy.**DataSelectionStrategy**(*trainloader*, *valloader*, *model*, *num_classes*, *linear_layer*, *loss*, *device*, *logger*)

Bases: `object`

Implementation of Data Selection Strategy class which serves as base class for other dataselectionstrategies for general learning frameworks. :param trainloader: Loading the training data using pytorch dataloader :type trainloader: class :param valloader: Loading the validation data using pytorch dataloader :type valloader: class :param model: Model architecture used for training :type model: class :param num_classes: Number of target classes in the dataset :type num_classes: int :param linear_layer: If True, we use the last fc layer weights and biases gradients

If False, we use the last fc layer biases gradients

Parameters

- **loss** (*class*) – PyTorch Loss function
- **device** (*str*) – The device being utilized - cpu | cuda
- **logger** (*class*) – logger object for logging the information

compute_gradients(*valid=False*, *perBatch=False*, *perClass=False*)

Computes the gradient of each element.

Here, the gradients are computed in a closed form using CrossEntropyLoss with reduction set to 'none'. This is done by calculating the gradients in last layer through addition of softmax layer.

Using different loss functions, the way we calculate the gradients will change.

For LogisticLoss we measure the Mean Absolute Error(MAE) between the pairs of observations. With reduction set to 'none', the loss is formulated as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = |x_n - y_n|,$$

where N is the batch size.

For MSELoss, we measure the Mean Square Error(MSE) between the pairs of observations. With reduction set to 'none', the loss is formulated as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. :param valid: if True, the function also computes the validation gradients :type valid: bool :param perBatch: if True, the function computes the gradients of each mini-batch :type perBatch: bool :param perClass: if True, the function computes the gradients using perclass dataloaders :type perClass: bool

get_labels(*valid=False*)

select(*budget*, *model_params*)

update_model(*model_params*)
Update the models parameters

Parameters **model_params** (*OrderedDict*) – Python dictionary object containing models parameters

GLISTER[Page 17, 1]

class cords.selectionstrategies.SL.glisterstrategy.**GLISTERStrategy**(*trainloader*, *valloader*, *model*, *loss_func*, *eta*, *device*, *num_classes*, *linear_layer*, *selection_type*, *greedy*, *logger*, *r=15*)

Bases: *cords.selectionstrategies.SL.dataselectionstrategy.DataSelectionStrategy*

Implementation of GLISTER-ONLINE Strategy from the paper[Page 17, 1] for supervised learning frameworks. GLISTER-ONLINE methods tries to solve the bi-level optimization problem given below:

$$\overbrace{\underset{S\subseteq\mathcal{U},|S|\leq k}{\operatorname{argmin}} L_V(\underbrace{\underset{\theta}{\operatorname{argmin}} L_T(\theta,S)}_{inner-level},\mathcal{V})}^{outer-level}$$

In the above equation, $\mathcal{U}$ denotes the training set, $\mathcal{V}$ denotes the validation set that guides the subset selection process, L_T denotes the training loss, L_V denotes the validation loss, S denotes the data subset selected at each round, and k is the budget for the subset.

Since, solving the complete inner-optimization is expensive, GLISTER-ONLINE adopts a online one-step meta approximation where we approximate the solution to inner problem by taking a single gradient step.

The optimization problem after the approximation is as follows:

$$\overbrace{\underset{S\subseteq\mathcal{U},|S|\leq k}{\operatorname{argmin}} L_V(\underbrace{\theta-\eta\nabla_\theta L_T(\theta,S)}_{inner-level},\mathcal{V})}^{outer-level}$$

In the above equation, η denotes the step-size used for one-step gradient update.

GLISTER-ONLINE also makes an additional approximation called Taylor-Series approximation to easily solve the outer problem using a greedy selection algorithm. The Taylor series approximation is as follows:

$$L_V(\theta-\eta\nabla_\theta L_T(\theta,S),\mathcal{V})\approx L_V(\theta)-\eta\nabla_\theta L_T(\theta,S)^T\nabla_\theta L_V(\theta,\mathcal{V})$$

The Optimization problem after the Taylor series approximation is as follows:

$$\underset{S\subseteq\mathcal{U},|S|\leq k}{\operatorname{argmin}} L_V(\theta-\eta\nabla_\theta L_T(\theta,S),\mathcal{V})\approx L_V(\theta)-\eta\nabla_\theta L_T(\theta,S)^T\nabla_\theta L_V(\theta,\mathcal{V})$$

Taylor's series approximation reduces the time complexity by reducing the need of calculating the validation loss for each element during greedy selection step which means reducing the number of forward passes required.

GLISTER-ONLINE is an adaptive subset selection algorithm that tries to select a subset every L epochs and the parameter L can be set in the original training loop.

[1] Krishnateja Killamsetty, Durga Sivasubramanian, Ganesh Ramakrishnan, and Rishabh Iyer. Glister: generalization based data subset selection for efficient and robust learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(9):8110–8118, May 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16988.

Parameters

trainloader: class Loading the training data using pytorch DataLoader

valloader: class Loading the validation data using pytorch DataLoader

model: class Model architecture used for training

loss_func: object Loss function object

eta: float Learning rate. Step size for the one step gradient update

device: str The device being utilized - cpu | cuda

num_classes: int The number of target classes in the dataset

linear_layer: bool If True, we use the last fc layer weights and biases gradients If False, we use the last fc layer biases gradients

selection_type: str Type of selection algorithm - - 'PerBatch' : PerBatch method is where GLISTER algorithm is applied on each minibatch data points. - 'PerClass' : PerClass method is where GLISTER algorithm is applied on each class data points seperately. - 'Supervised' : Supervised method is where GLISTER algorithm is applied on entire training data.

greedy: str Type of greedy selection algorithm - - 'RGreedy' : RGreedy Selection method is a variant of naive greedy where we just perform r rounds of greedy selection by choosing k/r points in each round. - 'Stochastic' : Stochastic greedy selection method is based on the algorithm presented in this paper[2] - 'Naive' : Normal naive greedy selection method that selects a single best element every step until the budget is fulfilled

logger: class logger class for logging the information

r [int, optional] Number of greedy selection rounds when selection method is RGreedy (default: 15)

`eval_taylor_modular`(*grads*)
Evaluate gradients

Parameters **grads** (*Tensor*) – Gradients

Returns **gains** – Matrix product of two tensors

Return type Tensor

`greedy_algo`(*budget*)

`select`(*budget*, *model_params*)
Apply naive greedy method for data selection

Parameters

- **budget** (*int*) – The number of data points to be selected
- **model_params** (*OrderedDict*) – Python dictionary object containing models parameters

Returns

- **greedySet** (*list*) – List containing indices of the best datapoints,
- **budget** (*Tensor*) – Tensor containing gradients of datapoints present in greedySet

[2] Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, Amin Karbasi, Jan Vondrak, and Andreas Krause. Lazier than lazy greedy. 2014. arXiv:1409.7938.

CRAIG[Page 19, 3]

class cords.selectionstrategies.SL.craigstrategy.**CRAIGStrategy**(*trainloader*, *valloader*, *model*, *loss*, *device*, *num_classes*, *linear_layer*, *if_convex*, *selection_type*, *logger*, *optimizer='lazy'*)

Bases: *cords.selectionstrategies.SL.dataselectionstrategy.DataSelectionStrategy*

Implementation of CRAIG Strategy from the paper[Page 19, 3] for supervised learning frameworks.

CRAIG strategy tries to solve the optimization problem given below for convex loss functions:

$$\sum_{i \in \mathcal{U}} \min_{j \in S, |S| \leq k} \|x^i - x^j\|$$

In the above equation, $\mathcal{U}$ denotes the training set where (x^i, y^i) denotes the i^{th} training data point and label respectively, L_T denotes the training loss, S denotes the data subset selected at each round, and k is the budget for the subset.

Since, the above optimization problem is not dependent on model parameters, we run the subset selection only once right before the start of the training.

CRAIG strategy tries to solve the optimization problem given below for non-convex loss functions:

$$\sum_{i \in \mathcal{U}} \min_{j \in S, |S| \leq k} \|\nabla_\theta {L_T}^i(\theta) - \nabla_\theta {L_T}^j(\theta)\|$$

In the above equation, $\mathcal{U}$ denotes the training set, L_T denotes the training loss, S denotes the data subset selected at each round, and k is the budget for the subset. In this case, CRAIG acts an adaptive subset selection strategy that selects a new subset every epoch.

Both the optimization problems given above are an instance of facility location problems which is a submodular function. Hence, it can be optimally solved using greedy selection methods.

Parameters

trainloader: class Loading the training data using pytorch DataLoader

valloader: class Loading the validation data using pytorch DataLoader

model: class Model architecture used for training

loss_type: class PyTorch Loss Function

device: str The device being utilized - cpu | cuda

num_classes: int The number of target classes in the dataset

linear_layer: bool Apply linear transformation to the data

if_convex: bool If convex or not

selection_type: str

Type of selection:

- 'PerClass': PerClass Implementation where the facility location problem is solved for each class seperately for speed ups.

[3] Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. Coresets for data-efficient training of machine learning models. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of Proceedings of Machine Learning Research, 6950–6960. PMLR, 13–18 Jul 2020. URL: https://proceedings.mlr.press/v119/mirzasoleiman20a.html.

- **'Supervised': Supervised Implementation where the facility location problem is solved using a sparse similar**
  assigning the similarity of a point with other points of different class to zero.
- 'PerBatch': PerBatch Implementation where the facility location problem tries to select subset of mini-batches.

logger [class]

- logger object for logging the information

optimizer: str Type of Greedy Algorithm

`compute_gamma`(*idxs*)
Compute the gamma values for the indices.

Parameters `idxs` (*list*) – The indices

Returns **gamma** – Gradient values of the input indices

Return type list

`compute_score`(*model_params*, *idxs*)
Compute the score of the indices.

Parameters

- `model_params` (*OrderedDict*) – Python dictionary object containing models parameters
- `idxs` (*list*) – The indices

`distance`(*x*, *y*, *exp=2*)
Compute the distance.

Parameters

- `x` (*Tensor*) – First input tensor
- `y` (*Tensor*) – Second input tensor
- `exp` (*float, optional*) – The exponent value (default: 2)

Returns **dist** – Output tensor

Return type Tensor

`get_similarity_kernel`()
Obtain the similarity kernel.

Returns **kernel** – Array of kernel values

Return type ndarray

`select`(*budget*, *model_params*)
Data selection method using different submodular optimization functions.

Parameters

- `budget` (*int*) – The number of data points to be selected
- `model_params` (*OrderedDict*) – Python dictionary object containing models parameters
- `optimizer` (*str*) – The optimization approach for data selection. Must be one of 'random', 'modular', 'naive', 'lazy', 'approximate-lazy', 'two-stage', 'stochastic', 'sample', 'greedi', 'bidirectional'

Returns

- **total_greedy_list** (*list*) – List containing indices of the best datapoints

- **gammas** (*list*) – List containing gradients of datapoints present in greedySet

GradMatch[Page 22, 4]

class cords.selectionstrategies.SL.gradmatchstrategy.**GradMatchStrategy**(*trainloader*, *valloader*, *model*, *loss*, *eta*, *device*, *num_classes*, *linear_layer*, *selection_type*, *logger*, *valid=False*, *v1=True*, *lam=0*, *eps=0.0001*)

Bases: *cords.selectionstrategies.SL.dataselectionstrategy.DataSelectionStrategy*

Implementation of GradMatch Strategy from the paper[Page 22, 4] for supervised learning frameworks.

GradMatch strategy tries to solve the optimization problem given below:

$$\min_{\mathbf{w},S:|S|\leq k} \|\sum_{i\in S} w_i \nabla_\theta L_T^i(\theta) - \nabla_\theta L(\theta)\|$$

In the above equation, $\mathbf{w}$ denotes the weight vector that contains the weights for each data instance, $\mathcal{U}$ training set where (x^i, y^i) denotes the i^{th} training data point and label respectively, L_T denotes the training loss, L denotes either training loss or validation loss depending on the parameter valid, S denotes the data subset selected at each round, and k is the budget for the subset.

The above optimization problem is solved using the Orthogonal Matching Pursuit(OMP) algorithm.

Parameters

trainloader: class Loading the training data using pytorch DataLoader

valloader: class Loading the validation data using pytorch DataLoader

model: class Model architecture used for training

loss: class PyTorch loss function for training

eta: float Learning rate. Step size for the one step gradient update

device: str The device being utilized - cpu | cuda

num_classes: int The number of target classes in the dataset

linear_layer: bool Apply linear transformation to the data

selection_type: str Type of selection - - 'PerClass': PerClass method is where OMP algorithm is applied on each class data points seperately. - 'PerBatch': PerBatch method is where OMP algorithm is applied on each minibatch data points. - 'PerClassPerGradient': PerClassPerGradient method is same as PerClass but we use the gradient corresponding to classification layer of that class only.

logger [class]

- logger object for logging the information

valid [bool] If valid==True, we use validation dataset gradient sum in OMP otherwise we use training dataset (default: False)

v1 [bool] If v1==True, we use newer version of OMP solver that is more accurate

[4] Krishnateja Killamsetty, Durga S, Ganesh Ramakrishnan, Abir De, and Rishabh Iyer. Grad-match: gradient matching based data subset selection for efficient deep model training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of Proceedings of Machine Learning Research, 5464–5474. PMLR, 18–24 Jul 2021. URL: https://proceedings.mlr.press/v139/killamsetty21a.html.

lam [float] Regularization constant of OMP solver

eps [float] Epsilon parameter to which the above optimization problem is solved using OMP algorithm

ompwrapper(*X*, *Y*, *bud*)

select(*budget*, *model_params*)
Apply OMP Algorithm for data selection

Parameters

- **budget** (*int*) – The number of data points to be selected
- **model_params** (*OrderedDict*) – Python dictionary object containing models parameters

Returns

- **idxs** (*list*) – List containing indices of the best datapoints,
- **gammas** (*weights tensors*) – Tensor containing weights of each instance

Random Strategy

class cords.selectionstrategies.SL.randomstrategy.**RandomStrategy**(*trainloader*, *online=False*)
Bases: object

This is the Random Selection Strategy class where we select a set of random points as a datasubset and often acts as baselines to compare other selection strategies.

Parameters **trainloader** (*class*) – Loading the training data using pytorch DataLoader

select(*budget*)
Perform random sampling of indices of size budget.

Parameters **budget** (*int*) – The number of data points to be selected

Returns

- **indices** (*ndarray*) – Array of indices of size budget selected randomly
- **gammas** (*Tensor*) – Gradient weight values of selected indices

Submodular Selection Strategy

class cords.selectionstrategies.SL.submodularselectionstrategy.**SubmodularSelectionStrategy**(*trainloader*, *valloader*, *model*, *loss*, *device*, *num_classes*, *linear_layer*, *if_convex*, *selection_type*, *submod_func_type*, *optimizer*)

Bases: *cords.selectionstrategies.SL.dataselectionstrategy.DataSelectionStrategy*

This class extends `selectionstrategies.supervisedlearning.dataselectionstrategy.DataSelectionStrategy` to include submodular optmization functions using apricot for data selection.

Parameters

- **trainloader** (*class*) – Loading the training data using pytorch DataLoader
- **valloader** (*class*) – Loading the validation data using pytorch DataLoader
- **model** (*class*) – Model architecture used for training
- **loss_type** (*class*) – The type of loss criterion
- **device** (*str*) – The device being utilized - cpu | cuda
- **num_classes** (*int*) – The number of target classes in the dataset
- **linear_layer** (*bool*) – Apply linear transformation to the data
- **if_convex** (*bool*) – If convex or not
- **selection_type** (*str*) – PerClass or Supervised
- **submod_func_type** (*str*) – The type of submodular optimization function. Must be one of 'facility-location', 'graph-cut', 'sum-redundancy', 'saturated-coverage'

compute_gamma(*idxs*)

Compute the gamma values for the indices.

Parameters **idxs** (*list*) – The indices

Returns **gamma** – Gradient values of the input indices

Return type list

compute_score(*model_params*, *idxs*)

Compute the score of the indices.

Parameters

- **model_params** (*OrderedDict*) – Python dictionary object containing models parameters
- **idxs** (*list*) – The indices

distance(*x*, *y*, *exp=2*)
Compute the distance.

Parameters

- **x** (*Tensor*) – First input tensor
- **y** (*Tensor*) – Second input tensor
- **exp** (*float, optional*) – The exponent value (default: 2)

Returns **dist** – Output tensor

Return type Tensor

get_similarity_kernel()
Obtain the similarity kernel.

Returns **kernel** – Array of kernel values

Return type ndarray

select(*budget*, *model_params*)
Data selection method using different submodular optimization functions.

Parameters

- **budget** (*int*) – The number of data points to be selected
- **model_params** (*OrderedDict*) – Python dictionary object containing models parameters
- **optimizer** (*str*) – The optimization approach for data selection. Must be one of 'random', 'modular', 'naive', 'lazy', 'approximate-lazy', 'two-stage', 'stochastic', 'sample', 'greedi', 'bidirectional'

Returns

- **total_greedy_list** (*list*) – List containing indices of the best datapoints
- **gammas** (*list*) – List containing gradients of datapoints present in greedySet

REFERENCES

Semi-supervised Learning Data Selection Strategies

In this section, we consider different data selection strategies geared towards efficient and robust learning in standard semi-supervised learning setting.

Data Selection Strategy - Base Class

class cords.selectionstrategies.SSL.dataselectionstrategy.**DataSelectionStrategy**(*trainloader*, *valloader*, *model*, *tea_model*, *ssl_alg*, *num_classes*, *linear_layer*, *loss*, *device*, *logger*)

Bases: object

Implementation of Data Selection Strategy class which serves as base class for other dataselectionstrategies for semi-supervised learning frameworks. :param trainloader: Loading the training data using pytorch dataloader :type trainloader: class :param valloader: Loading the validation data using pytorch dataloader :type valloader: class :param model: Model architecture used for training :type model: class :param tea_model: Teacher model architecture used for training :type tea_model: class :param ssl_alg: SSL algorithm class :type ssl_alg: class :param num_classes: Number of target classes in the dataset :type num_classes: int :param linear_layer: If True, we use the last fc layer weights and biases gradients

If False, we use the last fc layer biases gradients

Parameters

- **loss** (*class*) – Consistency loss function for unlabeled data with no reduction
- **device** (*str*) – The device being utilized - cpu | cuda
- **logger** (*class*) – logger file for printing the info

compute_gradients(*valid=False*, *perBatch=False*, *perClass=False*, *store_t=False*)

Computes the gradient of each element.

Here, the gradients are computed in a closed form using CrossEntropyLoss with reduction set to 'none'. This is done by calculating the gradients in last layer through addition of softmax layer.

Using different loss functions, the way we calculate the gradients will change.

For LogisticLoss we measure the Mean Absolute Error(MAE) between the pairs of observations. With reduction set to 'none', the loss is formulated as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = |x_n - y_n|,$$

where N is the batch size.

For MSELoss, we measure the Mean Square Error(MSE) between the pairs of observations. With reduction set to 'none', the loss is formulated as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. :param valid: if True, the function also computes the validation gradients :type valid: bool :param batch: if True, the function computes the gradients of each mini-batch :type batch: bool :param perClass: if True, the function computes the gradients using perclass dataloaders :type perClass: bool :param store_t: if True, the function stores the hypothesized weak augmentation targets and masks for unlabeled set. :type store_t: bool

get_labels(*valid=False*)

Function that iterates over labeled or unlabeled data and returns target or hypothesized labels.

Parameters **valid** (*bool*) – If True, iterate over the labeled set

select(*budget*, *model_params*, *tea_model_params*)
Abstract select function that is overloaded by the child classes

ssl_loss(*ul_weak_data*, *ul_strong_data*, *labels=False*)
Function that computes contrastive semi-supervised loss

Parameters

- **ul_weak_data** – Weak agumented version of unlabeled data
- **ul_strong_data** – Strong agumented version of unlabeled data
- **labels** (*bool*) – if labels, just return hypothesized labels of the unlabeled data

Returns

- **L_consistency** (*Consistency loss*)
- **y** (*Actual labels(Not used anywhere)*)
- **l1_strong** (*Penultimate layer outputs for strongly augmented version of unlabeled data*)
- **targets** (*Hypothesized labels*)
- **mask** (*mask vector of the unlabeled data*)

update_model(*model_params*, *tea_model_params*)
Update the models parameters

Parameters

- **model_params** (*OrderedDict*) – Python dictionary object containing model's parameters
- **tea_model_params** (*OrderedDict*) – Python dictionary object containing teacher model's parameters

RETRIEVE Strategy[1]

class cords.selectionstrategies.SSL.retrievestrategy.**RETRIEVEStrategy**(*trainloader*, *valloader*, *model*, *tea_model*, *ssl_alg*, *loss*, *eta*, *device*, *num_classes*, *linear_layer*, *selection_type*, *greedy*, *logger*, *r=15*, *valid=True*)

Bases: *cords.selectionstrategies.SSL.dataselectionstrategy.DataSelectionStrategy*

Implementation of RETRIEVE Strategy from the paper[1] for efficient and robust semi-supervised learning frameworks. RETRIEVE method tries to solve the bi-level optimization problem given below:

$$\overbrace{\mathcal{S}_t = \underset{\mathcal{S}\subseteq\mathcal{U}:|\mathcal{S}|\leq k}{\operatorname{argmin}} L_S\Big(\mathcal{D}, \underbrace{\underset{\theta}{\operatorname{argmin}}\left(L_S(\mathcal{D},\theta_t) + \lambda_t \sum_{j\in\mathcal{S}} \mathbf{m}_{jt} l_u(x_j,\theta_t)\right)}_{inner-level}\Big)}^{outer-level}$$

[1] Krishnateja Killamsetty, Xujiang Zhao, Feng Chen, and Rishabh K Iyer. RETRIEVE: coreset selection for efficient and robust semi-supervised learning. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*. 2021. URL: https://openreview.net/forum?id=jSz59N8NvUP.

Notation: Denote :math: *mathcal{D} = {x_i, y_i}_{i=1}^n* to be the labeled set with :math: *n* labeled data points, and :math: *mathcal{U} = {x_j}_{j=1}^m* to be the unlabeled set with :math: *m* data points. Let :math: *theta* be the classifier model parameters, :math: *l_s* be the labeled set loss function (such as cross-entropy loss) and :math: *l_u* be the unlabeled set loss, e.g. consistency-regularization loss, entropy loss, etc. Denote :math: *L_S(mathcal{D}, theta) = underset{i in mathcal{D}}{sum}l_{s}(theta, x_i, y_i)* and :math: *L_U(mathcal{U}, theta, mathbf{m}) = underset{j in mathcal{U}}{sum} mathbf{m}_i l_u(x_j, theta)* where :math: *mathbf{m} in {0, 1}^m* is the binary mask vector for unlabeled set. For notational convenience, we denote :math: *l_{si}(theta) = l_s(x_i, y_i, theta)* and denote :math: *$l_{uj}(theta) = l_u(x_j, theta)$*.

Since, solving the complete inner-optimization is expensive, RETRIEVE adopts a online one-step meta approximation where we approximate the solution to inner problem by taking a single gradient step.

The optimization problem after the approximation is as follows:

$$\mathcal{S}_t = \underset{\mathcal{S}\subseteq\mathcal{U}:|\mathcal{S}|\leq k}{\operatorname{argmin}} L_S(\mathcal{D}, \theta_t - \alpha_t \nabla_\theta L_S(\mathcal{D}, \theta_t) - \alpha_t \lambda_t \sum_{j\in\mathcal{S}} \mathbf{m}_{jt} \nabla_\theta l_u(x_j, \theta_t))$$

In the above equation, α_t denotes the step-size used for one-step gradient update.

RETRIEVE-ONLINE also makes an additional approximation called Taylor-Series approximation to easily solve the outer problem using a greedy selection algorithm. The Taylor series approximation is as follows:

$$L_S(\mathcal{D}, \theta_t - \alpha_t \nabla_\theta L_S(\mathcal{D}, \theta_t) - \alpha_t \lambda_t \sum_{j\in\mathcal{S}} \mathbf{m}_{jt} \nabla_\theta l_u(x_j, \theta_t)) \approx L_S(\mathcal{D}, \theta^S) - \alpha_t \lambda_t \nabla_\theta L_S(\mathcal{D}, \theta^S)^T \mathbf{m}_{et} \nabla_\theta l_u(x_e, \theta_t)$$

Taylor's series approximation reduces the time complexity by reducing the need of calculating the labeled set loss for each element during greedy selection step which means reducing the number of forward passes required.

RETRIEVE-ONLINE is an adaptive subset selection algorithm that tries to select a subset every L epochs and the parameter *L* can be set in the original training loop.

Parameters

trainloader: class Loading the training data using pytorch DataLoader

valloader: class Loading the validation data using pytorch DataLoader

model: class Model architecture used for training

tea_model: class Teacher model architecture used for training

ssl_alg: class SSL algorithm class

loss: class Consistency loss function for unlabeled data with no reduction

eta: float Learning rate. Step size for the one step gradient update

device: str The device being utilized - cpu | cuda

num_classes: int The number of target classes in the dataset

linear_layer: bool If True, we use the last fc layer weights and biases gradients If False, we use the last fc layer biases gradients

selection_type: str Type of selection algorithm - - 'PerBatch' : PerBatch method is where RETRIEVE algorithm is applied on each minibatch data points. - 'PerClass' : PerClass method is where RETRIEVE algorithm is applied on each class data points seperately. - 'Supervised' : Supervised method is where RETRIEVE algorithm is applied on entire training data.

greedy: str Type of greedy selection algorithm - - 'RGreedy' : RGreedy Selection method is a variant of naive greedy where we just perform r rounds of greedy selection by choosing k/r points in each round. - 'Stochas-

tic' : Stochastic greedy selection method is based on the algorithm presented in this paper[2] - 'Naive' : Normal naive greedy selection method that selects a single best element every step until the budget is fulfilled

logger: class Logger class for logging the information

r [int, optional] Number of greedy selection rounds when selection method is RGreedy (default: 15)

valid: bool

- If True, we select subset that maximizes the performance on the labeled set.
- If False, we select subset that maximizes the performance on the unlabeled set.

eval_taylor_modular(*grads*)
Evaluate gradients

Parameters **grads** (*Tensor*) – Gradients

Returns **gains** – Matrix product of two tensors

Return type Tensor

greedy_algo(*budget*)
Implement various greedy algorithms for data subset selection.

Parameters **budget** (*int*) – Budget of data points that needs to be sampled

select(*budget*, *model_params*, *tea_model_params*)
Apply naive greedy method for data selection

Parameters

- **budget** (*int*) – The number of data points to be selected
- **model_params** (*OrderedDict*) – Python dictionary object containing model's parameters
- **tea_model_params** (*OrderedDict*) – Python dictionary object containing teacher model's parameters

Returns

- **greedySet** (*list*) – List containing indices of the best datapoints,
- **budget** (*Tensor*) – Tensor containing gradients of datapoints present in greedySet

CRAIG Strategy[3]

class cords.selectionstrategies.SSL.craigstrategy.**CRAIGStrategy**(*trainloader*, *valloader*, *model*, *tea_model*, *ssl_alg*, *loss*, *device*, *num_classes*, *linear_layer*, *if_convex*, *selection_type*, *logger*, *optimizer='lazy'*)

Bases: *cords.selectionstrategies.SSL.dataselectionstrategy.DataSelectionStrategy*

Adapted Implementation of CRAIG Strategy from the paper[3] for semi-supervised learning setting.

[2] Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, Amin Karbasi, Jan Vondrak, and Andreas Krause. Lazier than lazy greedy. 2014. arXiv:1409.7938.

[3] Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. Coresets for data-efficient training of machine learning models. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of Proceedings of Machine Learning Research, 6950–6960. PMLR, 13–18 Jul 2020. URL: https://proceedings.mlr.press/v119/mirzasoleiman20a.html.

CRAIG strategy tries to solve the optimization problem given below for convex loss functions:

$$\sum_{i \in \mathcal{U}} \min_{j \in S, |S| \leq k} \|x^i - x^j\|$$

In the above equation, $\mathcal{U}$ denotes the training set where (x^i, y^i) denotes the i^{th} training data point and label respectively, L_T denotes the training loss, S denotes the data subset selected at each round, and k is the budget for the subset.

Since, the above optimization problem is not dependent on model parameters, we run the subset selection only once right before the start of the training.

CRAIG strategy tries to solve the optimization problem given below for non-convex loss functions:

$$\underset{\mathcal{S} \subseteq \mathcal{U}: |\mathcal{S}| \leq k}{\operatorname{argmin}} \sum_{i \in \mathcal{U}} \min_{j \in \mathcal{S}} \|\mathbf{m}_i \nabla_\theta l_u(x_i, \theta) - \mathbf{m}_j \nabla_\theta l_u(x_j, \theta)\|$$

In the above equation, $\mathcal{U}$ denotes the unlabeled set, l_u denotes the unlabeled loss, $\mathcal{S}$ denotes the data subset selected at each round, and k is the budget for the subset. In this case, CRAIG acts an adaptive subset selection strategy that selects a new subset every epoch.

Both the optimization problems given above are an instance of facility location problems which is a submodular function. Hence, it can be optimally solved using greedy selection methods.

Parameters

trainloader: class Loading the training data using pytorch DataLoader

valloader: class Loading the validation data using pytorch DataLoader

model: class Model architecture used for training

tea_model: class Teacher model architecture used for training

ssl_alg: class SSL algorithm class

loss: class Consistency loss function for unlabeled data with no reduction

device: str The device being utilized - cpu | cuda

num_classes: int The number of target classes in the dataset

linear_layer: bool Apply linear transformation to the data

if_convex: bool If convex or not

selection_type: str

Type of selection:

- 'PerClass': PerClass Implementation where the facility location problem is solved for each class seperately for speed ups.
- 'Supervised': Supervised Implementation where the facility location problem is solved using a sparse similarity matrix by assigning the similarity of a point with other points of different class to zero.
- 'PerBatch': PerBatch Implementation where the facility location problem tries to select subset of mini-batches.

logger: class Logger class for logging the information

optimizer: str Type of Greedy Algorithm

`compute_gamma`(*idxs*)

Compute the gamma values for the indices.

Parameters **idxs** (*list*) – The indices

Returns **gamma** – Gradient values of the input indices

Return type list

compute_score(*model_params*, *tea_model_params*, *idxs*)
Compute the score of the indices.

Parameters

- **model_params** (*OrderedDict*) – Python dictionary object containing model's parameters
- **tea_model_params** (*OrderedDict*) – Python dictionary object containing teacher model's parameters
- **idxs** (*list*) – The indices

distance(*x*, *y*, *exp=2*)
Compute the distance.

Parameters

- **x** (*Tensor*) – First input tensor
- **y** (*Tensor*) – Second input tensor
- **exp** (*float, optional*) – The exponent value (default: 2)

Returns **dist** – Output tensor

Return type Tensor

get_similarity_kernel()
Obtain the similarity kernel.

Returns **kernel** – Array of kernel values

Return type ndarray

select(*budget*, *model_params*, *tea_model_params*)
Data selection method using different submodular optimization functions.

Parameters

- **budget** (*int*) – The number of data points to be selected
- **model_params** (*OrderedDict*) – Python dictionary object containing models parameters
- **optimizer** (*str*) – The optimization approach for data selection. Must be one of 'random', 'modular', 'naive', 'lazy', 'approximate-lazy', 'two-stage', 'stochastic', 'sample', 'greedi', 'bidirectional'

Returns

- **total_greedy_list** (*list*) – List containing indices of the best datapoints
- **gammas** (*list*) – List containing gradients of datapoints present in greedySet

GradMatch Strategy[4]

class cords.selectionstrategies.SSL.gradmatchstrategy.**GradMatchStrategy**(*trainloader*, *valloader*, *model*, *tea_model*, *ssl_alg*, *loss*, *eta*, *device*, *num_classes*, *linear_layer*, *selection_type*, *logger*, *valid=False*, *v1=True*, *lam=0*, *eps=0.0001*)

Bases: *cords.selectionstrategies.SSL.dataselectionstrategy.DataSelectionStrategy*

Implementation of OMPGradMatch Strategy from the paper[4] for supervised learning frameworks.

OMPGradMatch strategy tries to solve the optimization problem given below:

$$\underset{\mathcal{S}\subseteq\mathcal{U}:|\mathcal{S}|\leq k,\{\mathbf{w}_j\}_{j\in[1,|\mathcal{S}|]}:\forall_j \mathbf{w}_j\geq 0}{\operatorname{argmin}} \left\| \sum_{i\in\mathcal{U}} \mathbf{m}_i \nabla_\theta l_u(x_i,\theta) - \sum_{j\in\mathcal{S}} \mathbf{m}_j \mathbf{w}_j \nabla_\theta l_u(x_j,\theta) \right\|$$

In the above equation, $\mathbf{w}$ denotes the weight vector that contains the weights for each data instance, $\mathcal{U}$ denotes the unlabeled set where (x^i, y^i) denotes the i^{th} training data point and label respectively, l_u denotes the unlabeled loss, $\mathcal{S}$ denotes the data subset selected at each round, and k is the budget for the subset.

The above optimization problem is solved using the Orthogonal Matching Pursuit(OMP) algorithm.

Parameters

trainloader: class Loading the training data using pytorch DataLoader

valloader: class Loading the validation data using pytorch DataLoader

model: class Model architecture used for training

tea_model: class Teacher model architecture used for training

ssl_alg: class SSL algorithm class

loss: class Consistency loss function for unlabeled data with no reduction

eta: float Learning rate. Step size for the one step gradient update

device: str The device being utilized - cpu | cuda

num_classes: int The number of target classes in the dataset

linear_layer: bool Apply linear transformation to the data

selection_type: str Type of selection - - 'PerClass': PerClass method is where OMP algorithm is applied on each class data points seperately. - 'PerBatch': PerBatch method is where OMP algorithm is applied on each minibatch data points. - 'PerClassPerGradient': PerClassPerGradient method is same as PerClass but we use the gradient corresponding to classification layer of that class only.

logger [class] logger file for printing the info

valid [bool, optional] If valid==True we use validation dataset gradient sum in OMP otherwise we use training dataset (default: False)

v1 [bool] If v1==True, we use newer version of OMP solver that is more accurate

[4] Krishnateja Killamsetty, Durga S, Ganesh Ramakrishnan, Abir De, and Rishabh Iyer. Grad-match: gradient matching based data subset selection for efficient deep model training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of Proceedings of Machine Learning Research, 5464–5474. PMLR, 18–24 Jul 2021. URL: https://proceedings.mlr.press/v139/killamsetty21a.html.

lam [float] Regularization constant of OMP solver

eps [float] Epsilon parameter to which the above optimization problem is solved using OMP algorithm

ompwrapper(*X*, *Y*, *bud*)
Wrapper function that instantiates the OMP algorithm

Parameters

X: Individual datapoint gradients

Y: Gradient sum that needs to be matched to.

bud: Budget of datapoints that needs to be sampled from the unlabeled set

Returns

- **idxs** (*list*) – List containing indices of the best datapoints,
- **gammas** (*weights tensors*) – Tensor containing weights of each instance

select(*budget*, *model_params*, *tea_model_params*)
Apply OMP Algorithm for data selection

Parameters

- **budget** (*int*) – The number of data points to be selected
- **model_params** (*OrderedDict*) – Python dictionary object containing model's parameters
- **tea_model_params** (*OrderedDict*) – Python dictionary object containing teacher model's parameters

Returns

- **idxs** (*list*) – List containing indices of the best datapoints,
- **gammas** (*weights tensors*) – Tensor containing weights of each instance

Random Strategy

class cords.selectionstrategies.SSL.randomstrategy.**RandomStrategy**(*trainloader*, *online=False*)
Bases: object

This is the Random Selection Strategy class where we select a set of random points as a datasubset and often acts as baselines to compare other subset selection strategies.

Parameters **trainloader** (*class*) – Loading the training data using pytorch DataLoader

select(*budget*)
Perform random sampling of indices of size budget.

Parameters **budget** (*int*) – The number of data points to be selected

Returns

- **indices** (*ndarray*) – Array of indices of size budget selected randomly
- **gammas** (*Tensor*) – Gradient weight values of selected indices

REFERENCES

1.1.2 Subset Selection Dataloaders

Essentially, with subset selection-based data loaders, it is pretty straightforward to use subset selection strategies directly because they are integrated directly into subset data loaders; this allows users to use subset selection strategies directly by using their respective subset selection data loaders.

Below is an example that shows the subset selection process is simplified by just calling a data loader in supervised learning setting,

```
dss_args = dict(model=model,
                loss=criterion_nored,
                eta=0.01,
                num_classes=10,
                num_epochs=300,
                device='cuda',
                fraction=0.1,
                select_every=20,
                kappa=0,
                linear_layer=False,
                selection_type='SL',
                greedy='Stochastic')
dss_args = DotMap(dss_args)

dataloader = GLISTERDataLoader(trainloader, valloader, dss_args, logger,
                               batch_size=20,
                               shuffle=True,
                               pin_memory=False)

for epoch in range(num_epochs):
    for _, (inputs, targets, weights) in enumerate(dataloader):
        """
        Standard PyTorch training loop using weighted loss

        Our training loop differs from the standard PyTorch training loop in that along␣
↪with
        data samples and their associated target labels; we also have additional sample␣
↪weight
        information from the subset data loader, which can be used to calculate the␣
↪weighted
        loss for gradient descent. We can calculate the weighted loss by using default␣
↪PyTorch
        loss functions with no reduction as follows:
        """
        # Convert inputs, targets, and weights to the required device
        inputs = inputs.to(self.cfg.train_args.device)
        targets = targets.to(self.cfg.train_args.device, non_blocking=True)
        weights = weights.to(self.cfg.train_args.device)

        # Zero the optimizer gradients to prevent gradient accumulation
```

(continues on next page)

(continued from previous page)

```
        optimizer.zero_grad()

        #Model forward pass over the inputs
        outputs = model(inputs)

        # Get individual sample losses with no reduction
        losses = criterion_nored(outputs, targets)

        # Get weighted loss by a dotproduct of the losses vector with sample weights
        loss = torch.dot(losses, weights / (weights.sum()))

        # Do backprop on the weighted loss
        loss.backward()

        # Step the model based on the gradient values
        optimizer.step()
```

In our current version, we deployed subset selection data loaders in supervised learning and semi-supervised learning settings.

Supervised Learning Subset Selection Data Loaders

In this section, we consider different subset selection based data loaders geared towards efficient and robust learning in standard supervised learning setting.

DSS Dataloader (Base Class)

class cords.utils.data.dataloader.SL.dssdataloader.**DSSDataLoader**(*full_data*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: `object`

Implementation of DSSDataLoader class which serves as base class for dataloaders of other selection strategies for supervised learning framework.

Parameters

- **full_data** (*torch.utils.data.Dataset Class*) – Full dataset from which data subset needs to be selected.
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger class for logging the information

Non-Adaptive subset selection Data Loaders

class cords.utils.data.dataloader.SL.nonadaptive.nonadaptivedataloader.**NonAdaptiveDSSDataLoader**(*tra*
val
dss
log
ger
**ar*
***k*

Bases: *cords.utils.data.dataloader.SL.dssdataloader.DSSDataLoader*

Implementation of NonAdaptiveDSSDataLoader class which serves as base class for dataloaders of other nonadaptive subset selection strategies for supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SL.nonadaptive.craigdataloader.**CRAIGDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.nonadaptive.nonadaptivedataloader.NonAdaptiveDSSDataLoader*

Implements of CRAIGDataLoader that serves as the dataloader for the nonadaptive CRAIG subset selection strategy from the paper[1].

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for CRAIG subset selection strategy
- **logger** (*class*) – Logger for logging the information

[1] Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. Coresets for data-efficient training of machine learning models. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of Proceedings of Machine Learning Research, 6950–6960. PMLR, 13–18 Jul 2020. URL: https://proceedings.mlr.press/v119/mirzasoleiman20a.html.

class cords.utils.data.dataloader.SL.nonadaptive.submoddataloader.**FacLocDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.nonadaptive.submoddataloader.SubmodDataLoader*

Implementation of FacLocDataLoader class for the nonadaptive facility location based subset selection strategy for supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SL.nonadaptive.submoddataloader.**GraphCutDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.nonadaptive.submoddataloader.SubmodDataLoader*

Implementation of GraphCutDataLoader class for the nonadaptive graph cut function based subset selection strategy for supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SL.nonadaptive.submoddataloader.**SaturatedCoverageDataLoader**(*train_*
val_lo
dss_a
log-
ger,
**args*
***kwa*

Bases: *cords.utils.data.dataloader.SL.nonadaptive.submoddataloader.SubmodDataLoader*

Implementation of SaturatedCoverageDataLoader class for the nonadaptive saturated coverage function based subset selection strategy for supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SL.nonadaptive.submoddataloader.**SubmodDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.nonadaptive.nonadaptivedataloader.NonAdaptiveDSSDataLoader*

Implementation of SubmodDataLoader class for the nonadaptive submodular subset selection strategies for supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SL.nonadaptive.submoddataloader.**SumRedundancyDataLoader**(*train_loade*
val_loader, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.nonadaptive.submoddataloader.SubmodDataLoader*

Implementation of SumRedundancyDataLoader class for the nonadaptive sum redundancy function based subset selection strategy for supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary

- **logger** (*class*) – Logger for logging the information

Adaptive subset selection Data Loaders

class cords.utils.data.dataloader.SL.adaptive.adaptivedataloader.**AdaptiveDSSDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.dssdataloader.DSSDataLoader*

Implementation of AdaptiveDSSDataLoader class which serves as base class for dataloaders of other adaptive subset selection strategies for supervised learning framework.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

resample()

Function that resamples the subset indices and recalculates the subset weights

class cords.utils.data.dataloader.SL.adaptive.glisterdataloader.**GLISTERDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.adaptive.adaptivedataloader.AdaptiveDSSDataLoader*

Implements of GLISTERDataLoader that serves as the dataloader for the adaptive GLISTER subset selection strategy from the paper[2].

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for GLISTER subset selection strategy
- **logger** (*class*) – Logger for logging the information

[2] Krishnateja Killamsetty, Durga Sivasubramanian, Ganesh Ramakrishnan, and Rishabh Iyer. Glister: generalization based data subset selection for efficient and robust learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(9):8110–8118, May 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16988.

class cords.utils.data.dataloader.SL.adaptive.craigdataloader.**CRAIGDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.adaptive.adaptivedataloader.AdaptiveDSSDataLoader*

Implements of CRAIGDataLoader that serves as the dataloader for the adaptive CRAIG subset selection strategy from the paper[Page 17, 1].

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for CRAIG subset selection strategy
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SL.adaptive.gradmatchdataloader.**GradMatchDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.adaptive.adaptivedataloader.AdaptiveDSSDataLoader*

Implements of GradMatchDataLoader that serves as the dataloader for the adaptive GradMatch subset selection strategy from the paper[3].

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for GradMatch subset selection strategy
- **logger** (*class*) – Logger for logging the information

[3] Krishnateja Killamsetty, Durga S, Ganesh Ramakrishnan, Abir De, and Rishabh Iyer. Grad-match: gradient matching based data subset selection for efficient deep model training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of Proceedings of Machine Learning Research, 5464–5474. PMLR, 18–24 Jul 2021. URL: https://proceedings.mlr.press/v139/killamsetty21a.html.

class cords.utils.data.dataloader.SL.adaptive.randomdataloader.**RandomDataLoader**(*train_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.adaptive.adaptivedataloader.AdaptiveDSSDataLoader*

Implements of RandomDataLoader that serves as the dataloader for the non-adaptive Random subset selection strategy.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for Random subset selection strategy
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SL.adaptive.olrandomdataloader.**OLRandomDataLoader**(*train_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SL.adaptive.adaptivedataloader.AdaptiveDSSDataLoader*

Implements of OLRandomDataLoader that serves as the dataloader for the adaptive Random subset selection strategy.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for Random subset selection strategy
- **logger** (*class*) – Logger for logging the information

REFERENCES

Semi-supervised Learning Subset Selection Data Loaders

In this section, we consider different subset selection based data loaders geared towards efficient and robust learning in standard semi-supervised learning setting.

DSS Dataloader (Base Class)

class cords.utils.data.dataloader.SSL.dssdataloader.**DSSDataLoader**(*full_data*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: object

Implementation of DSSDataLoader class which serves as base class for dataloaders of other selection strategies for semi-supervised learning framework.

Parameters

- **full_data** (*torch.utils.data.Dataset Class*) – Full dataset from which data subset needs to be selected.
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger class for logging the information

Non-Adaptive subset selection Data Loaders

class cords.utils.data.dataloader.SSL.nonadaptive.nonadaptivedataloader.**NonAdaptiveDSSDataLoader**(*tr va ds lo ge * ***

Bases: *cords.utils.data.dataloader.SSL.dssdataloader.DSSDataLoader*

Implementation of NonAdaptiveDSSDataLoader class which serves as base class for dataloaders of other nonadaptive subset selection strategies for semi-supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SSL.nonadaptive.craigdataloader.**CRAIGDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SSL.nonadaptive.nonadaptivedataloader.NonAdaptiveDSSDataLoader*

Implements of CRAIGDataLoader that serves as the dataloader for the nonadaptive CRAIG subset selection strategy for semi-supervised learning and is an adapted version from the paper[1].

[1] Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. Coresets for data-efficient training of machine learning models. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of Proceedings of Machine Learning Research, 6950–6960. PMLR, 13–18 Jul 2020. URL: https://proceedings.mlr.press/v119/mirzasoleiman20a.html.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for CRAIG subset selection strategy
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SSL.nonadaptive.submoddataloader.**FacLocDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SSL.nonadaptive.submoddataloader.SubmodDataLoader*

Implementation of FacLocDataLoader class for the nonadaptive facility location based subset selection strategy for semi-supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SSL.nonadaptive.submoddataloader.**GraphCutDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SSL.nonadaptive.submoddataloader.SubmodDataLoader*

Implementation of GraphCutDataLoader class for the nonadaptive graph cut function based subset selection strategy for semi-supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SSL.nonadaptive.submoddataloader.**SaturatedCoverageDataLoader**(*train*
*val_*
*dss_*
log-
ger,
**arg*
***kw*

Bases: *cords.utils.data.dataloader.SSL.nonadaptive.submoddataloader.SubmodDataLoader*

Implementation of SaturatedCoverageDataLoader class for the nonadaptive saturated coverage function based subset selection strategy for semi-supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SSL.nonadaptive.submoddataloader.**SubmodDataLoader**(*train_loader*, *val_loader*, *dss_args*, *log-*
ger,
**args*,
***kwargs*)

Bases: *cords.utils.data.dataloader.SSL.nonadaptive.nonadaptivedataloader.NonAdaptiveDSSDataLoader*

Implementation of SubmodDataLoader class for the nonadaptive submodular subset selection strategies for semi-supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SSL.nonadaptive.submoddataloader.**SumRedundancyDataLoader**(*train_loa*
val_loade
dss_args,
log-
ger,
**args*,
***kwargs*

Bases: *cords.utils.data.dataloader.SSL.nonadaptive.submoddataloader.SubmodDataLoader*

Implementation of SumRedundancyDataLoader class for the nonadaptive sum redundancy function based subset selection strategy for semi-supervised learning setting.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

Adaptive subset selection Data Loaders

class cords.utils.data.dataloader.SSL.adaptive.adaptivedataloader.**AdaptiveDSSDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SSL.dssdataloader.DSSDataLoader*

Implementation of AdaptiveDSSDataLoader class which serves as base class for dataloaders of other adaptive subset selection strategies for semi-supervised learning framework.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary
- **logger** (*class*) – Logger for logging the information

resample()
Function that resamples the subset indices and recalculates the subset weights

class cords.utils.data.dataloader.SSL.adaptive.retrievedataloader.**RETRIEVEDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SSL.adaptive.adaptivedataloader.AdaptiveDSSDataLoader*

Implements of RETRIEVEDataLoader that serves as the dataloader for the adaptive RETRIEVE subset selection strategy from the paper[2].

[2] Krishnateja Killamsetty, Xujiang Zhao, Feng Chen, and Rishabh K Iyer. RETRIEVE: coreset selection for efficient and robust semi-supervised learning. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*. 2021. URL: https://openreview.net/forum?id=jSz59N8NvUP.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for GLISTER subset selection strategy
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SSL.adaptive.craigdataloader.**CRAIGDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SSL.adaptive.adaptivedataloader.AdaptiveDSSDataLoader*

Implements of CRAIGDataLoader that serves as the dataloader for the adaptive CRAIG subset selection strategy for semi-supervised learning and is an adapted version from the paper[Page 17, 1].

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **val_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the validation dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for CRAIG subset selection strategy
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SSL.adaptive.gradmatchdataloader.**GradMatchDataLoader**(*train_loader*, *val_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SSL.adaptive.adaptivedataloader.AdaptiveDSSDataLoader*

Implements of GradMatchDataLoader that serves as the dataloader for the adaptive GradMatch subset selection strategy for semi-supervised learning and is an adapted version of the one given in the paper[3]. :param train_loader: Dataloader of the training dataset :type train_loader: torch.utils.data.DataLoader class :param val_loader: Dataloader of the validation dataset :type val_loader: torch.utils.data.DataLoader class :param dss_args: Data subset selection arguments dictionary required for GradMatch subset selection strategy :type dss_args: dict :param logger: Logger for logging the information :type logger: class

[3] Krishnateja Killamsetty, Durga S, Ganesh Ramakrishnan, Abir De, and Rishabh Iyer. Grad-match: gradient matching based data subset selection for efficient deep model training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of Proceedings of Machine Learning Research, 5464–5474. PMLR, 18–24 Jul 2021. URL: https://proceedings.mlr.press/v139/killamsetty21a.html.

class cords.utils.data.dataloader.SSL.adaptive.randomdataloader.**RandomDataLoader**(*train_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SSL.adaptive.adaptivedataloader.AdaptiveDSSDataLoader*

Implements of RandomDataLoader that serves as the dataloader for the non-adaptive Random subset selection strategy.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for Random subset selection strategy
- **logger** (*class*) – Logger for logging the information

class cords.utils.data.dataloader.SSL.adaptive.olrandomdataloader.**OLRandomDataLoader**(*train_loader*, *dss_args*, *logger*, **args*, ***kwargs*)

Bases: *cords.utils.data.dataloader.SSL.adaptive.adaptivedataloader.AdaptiveDSSDataLoader*

Implements of OLRandomDataLoader that serves as the dataloader for the adaptive Random subset selection strategy.

Parameters

- **train_loader** (*torch.utils.data.DataLoader class*) – Dataloader of the training dataset
- **dss_args** (*dict*) – Data subset selection arguments dictionary required for Random subset selection strategy
- **logger** (*class*) – Logger for logging the information

REFERENCES

1.1.3 Datasets

We have added functionality to load several existing datasets for both supervised and semi-supervised learning settings. Use the gen_dataset function in the file cords.utils.datasets.SL.builder for loading the dataset in the supervised learning setting. Similarly, use the gen_dataset function in the file cords.utils.datasets.SSL.builder for loading the dataset in the semi-supervised learning setting.

In the Supervised learning setting, below given is a list of datasets supported:

- dna

- boston
- adult
- connect_4
- letter
- satimage
- svmguide1
- usps
- ijcnn1
- sklearn-digits
- covertype
- census
- mnist
- fashion-mnist
- cifar10
- cifar100
- svhn
- kmnist
- stl10
- emnist
- celeba

In the Semi-supervised learning setting, below given is a list of datasets supported:

- svhn
- stl10
- cifar10
- cifar100
- cifarOOD
- mnistOOD
- cifarImbalance

Dataset generator for Supervised Learning

In this section, we consider the dataset generator function for performing experiments in both standard and robust supervised learning scenarios.

Dataset Builder

class cords.utils.data.datasets.SL.builder.**CustomDataset**(*data*, *target*, *device=None*, *transform=None*, *isreg=False*)

Bases: torch.utils.data.dataset.Dataset

class cords.utils.data.datasets.SL.builder.**CustomDataset_WithId**(*data*, *target*, *transform=None*)

Bases: torch.utils.data.dataset.Dataset

class cords.utils.data.datasets.SL.builder.**GlueDataset**(*glue_dataset*, *sentence_str*, *label_str*, *clean_type*, *num_classes*, *wordvec_dim*, *wordvec*, *device='cpu'*)

Bases: torch.utils.data.dataset.Dataset

class cords.utils.data.datasets.SL.builder.**SSTDataset**(*path_to_dataset*, *name*, *num_classes*, *wordvec_dim*, *wordvec*, *device='cpu'*)

Bases: torch.utils.data.dataset.Dataset

label_tmp = None

class cords.utils.data.datasets.SL.builder.**Trec6Dataset**(*data_path*, *cls_to_num*, *num_classes*, *wordvec_dim*, *wordvec*, *device='cpu'*)

Bases: torch.utils.data.dataset.Dataset

cords.utils.data.datasets.SL.builder.**census_load**(*path*, *dim*, *save_data=False*)

cords.utils.data.datasets.SL.builder.**clean_data**(*sentence*, *type=0*, *TREC=False*)

cords.utils.data.datasets.SL.builder.**clean_lawschool_full**(*path*)

cords.utils.data.datasets.SL.builder.**create_imbalance**(*x_trn*, *y_trn*, *x_val*, *y_val*, *x_tst*, *y_tst*, *num_cls*, *ratio*)

cords.utils.data.datasets.SL.builder.**create_noisy**(*y_trn*, *num_cls*, *noise_ratio=0.8*)

cords.utils.data.datasets.SL.builder.**csv_file_load**(*path*, *dim*, *save_data=False*)

cords.utils.data.datasets.SL.builder.**gen_dataset**(*datadir*, *dset_name*, *feature*, *isnumpy=False*, ***kwargs*)

Generate train, val, and test datasets for supervised learning setting.

Parameters

- **datadir** (*str*) – Dataset directory in which the data is present or needs to be downloaded.
- **dset_name** (*str*) – dataset name, [‘cifar10’, ‘cifar100’, ‘svhn’, ‘stl10’]
- **feature** (*str*) –

  if ‘classimb’, generate datasets wth class imbalance

  – Needs keyword argument ‘classimb_ratio’

  elif ‘noise’, generate datasets with label noise otherwise, generate standard datasets
- **isnumpy** (*bool*) – if True, return datasets in numpy format instead of tensor format

cords.utils.data.datasets.SL.builder.**get_class**(*sentiment*, *num_classes*)

cords.utils.data.datasets.SL.builder.**libsvm_file_load**(*path*, *dim*, *save_data=False*)

cords.utils.data.datasets.SL.builder.**loadGloveModel**(*gloveFile*)

class cords.utils.data.datasets.SL.builder.**standard_scaling**

Bases: object

fit_transform(*data*)

transform(*data*)

Dataset generator for Semi-supervised Learning

In this section, we consider the dataset generator function for performing experiments in both standard and robust semi-supervised learning scenarios.

Dataset Builder

cords.utils.data.datasets.SSL.builder.**gen_dataloader**(*root*, *dataset*, *validation_split*, *cfg*, *logger=None*)

generate train, val, and test dataloaders

Parameters

- **root** (*str*) – root directory
- **dataset** (*str*) – dataset name, ['cifar10', 'cifar100', 'svhn', 'stl10']
- **validation_split** (*bool*) – if True, return validation loader. validation data is made from training data
- **cfg** (*argparse.Namespace or something*) –
- **logger** (*logging.Logger*) –

cords.utils.data.datasets.SSL.builder.**gen_dataset**(*root*, *dataset*, *validation_split*, *cfg*, *logger=None*)

generate train, val, and test datasets

Parameters

- **root** (*str*) – root directory in which data is present or needs to be downloaded
- **dataset** (*str*) – dataset name, Existing dataset choices: ['cifar10', 'cifar100', 'svhn', 'stl10', 'cifarOOD', 'mnistOOD', 'cifarImbalance']
- **validation_split** (*bool*) – if True, return validation loader. We use 10% random split of training data as validation data
- **cfg** (*argparse.Namespace or dict*) – Dictionary containing necessary arguments for generating the dataset
- **logger** (*logging.Logger*) – Logger class for logging the information

1.1.4 Models

We have incorporated several neural network architectures in the CORDS repository in the cords.utils.models folder. Below given is a list of Neural network architectures that are currently supported by CORDS:

- densenet
- dla
- dla_simple
- dpn
- efficientnet

- googlenet
- lenet
- mobilenet
- mobilenetv2
- pnasnet
- preact_resnet
- regnet
- resnet
- resnext
- senet
- shufflenet
- shufflenetv2
- vgg

1.1.5 Configuration files of CORDS for Training

This page gives a tutorial on how to generate your custom training configuration files for SL, SSL and HPO. Configuration files can be used to select datasets, training configuration, and subset selection strategy settings. These configuration files can be in python dictionary or yaml format. However, for the sake of simplicity we show config files in python.

Configuration files for SL

```
config = dict(setting="SL",
        is_reg = False,
        dataset=dict(name="cifar10",
                     datadir="../data",
                     feature="dss",
                     type="image"),

        dataloader=dict(shuffle=True,
                        batch_size=20,
                        pin_memory=True),

        model=dict(architecture='ResNet18',
                     type='pre-defined',
                     numclasses=10),

        ckpt=dict(is_load=False,
                     is_save=True,
                     dir='results/',
                     save_every=20),

        loss=dict(type='CrossEntropyLoss',
                     use_sigmoid=False),
```

(continues on next page)

(continued from previous page)

```
        optimizer=dict(type="sgd",
                       momentum=0.9,
                       lr=0.01,
                       weight_decay=5e-4),

        scheduler=dict(type="cosine_annealing",
                       T_max=300),

        dss_args=dict(type="CRAIG",
                           fraction=0.1,
                           select_every=20,
                           kappa=0,
                           linear_layer=False,
                           optimizer='lazy',
                           selection_type='PerClass'
                           ),

        train_args=dict(num_epochs=300,
                       device="cuda",
                       print_every=10,
                       results_dir='results/',
                       print_args=["val_loss", "val_acc", "tst_loss", "tst_acc
↪", "time"],
                       return_args=[]
                       )
        )
```

The SL configuration files consists of following sections: #. Dataset(dataset) #. Data Loader(dataloader) #. Checkpoint Arguments (ckpt) #. Training Loss (loss) #. Training Optimizer (optimizer) #. Training Scheduler (scheduler) #. Data subset selection Arguments (dss_args) #. Training Arguments (train_args)

You can refer to various configuration examples in the configs/ folders of the CORDS repository.

Configuration files for SSL

```
# Learning setting
config = dict(setting="SSL",
      dataset=dict(name="cifar10",
                   root="../data",
                   feature="dss",
                   type="pre-defined",
                   num_labels=4000,
                   val_ratio=0.1,
                   ood_ratio=0.5,
                   random_split=False,
                   whiten=False,
                   zca=True,
                   labeled_aug='WA',
                   unlabeled_aug='WA',
                   wa='t.t.f',
```

(continues on next page)

(continued from previous page)

```
                strong_aug=False),

    dataloader=dict(shuffle=True,
                    pin_memory=True,
                    num_workers=8,
                    l_batch_size=50,
                    ul_batch_size=50),

    model=dict(architecture='wrn',
               type='pre-defined',
               numclasses=10),

    ckpt=dict(is_load=False,
              is_save=True,
              checkpoint_model='model.ckpt',
              checkpoint_optimizer='optimizer.ckpt',
              start_iter=None,
              checkpoint=10000),

    loss=dict(type='CrossEntropyLoss',
              use_sigmoid=False),

    optimizer=dict(type="sgd",
                   momentum=0.9,
                   lr=0.03,
                   weight_decay=0,
                   nesterov=True,
                   tsa=False,
                   tsa_schedule='linear'),

    scheduler=dict(lr_decay="cos",
                   warmup_iter=0),

    ssl_args=dict(alg='vat',
                  coef=0.3,
                  ema_teacher=False,
                  ema_teacher_warmup=False,
                  ema_teacher_factor=0.999,
                  ema_apply_wd=False,
                  em=0,
                  threshold=None,
                  sharpen=None,
                  temp_softmax=None,
                  consis='ce',
                  eps=6,
                  xi=1e-6,
                  vat_iter=1
                  ),

    ssl_eval_args=dict(weight_average=False,
                       wa_ema_factor=0.999,
                       wa_apply_wd=False),
```

(continues on next page)

(continued from previous page)

```
    dss_args=dict(type="RETRIEVE",
                  fraction=0.1,
                  select_every=20,
                  kappa=0,
                  linear_layer=False,
                  selection_type='Supervised',
                  greedy='Stochastic',
                  valid=True),

    train_args=dict(iteration=500000,
                    max_iter=-1,
                    device="cuda",
                    results_dir='results/',
                    disp=256,
                    seed=96)
    )
```

Configuration files for HPO

```
from ray import tune

config = dict(setting= "hyperparamtuning",
    # parameter for subset selection
    # all settings for subset selection will be fetched from here
    subset_config = "configs/SL/config_gradmatchpb-warm_cifar100.py",
    # parameters for hyper-parameter tuning
    # search space for hyper-parameter tuning
    space = dict(
            learning_rate=tune.uniform(0.001, 0.01),
            learning_rate1=tune.uniform(0.001, 0.01),
            learning_rate2=tune.uniform(0.001, 0.01),
            learning_rate3=tune.uniform(0.001, 0.01),
            scheduler= tune.choice(['cosine_annealing', 'linear_decay']),
            nesterov= tune.choice([True, False]),
            gamma= tune.uniform(0.05, 0.5),
            ),

    # tuning algorithm
    search_algo = "TPE",

    # number of hyper-parameter set to try
    num_evals = 27,

    # metric to be optimized, for 'mean_loss' metric mode should be 'min'
    metric = "mean_accuracy",
    mode = "max",

    # scheduler to be used (i.e ASHAScheduler)
    # scheduler terminates trials that perform poorly
```

(continues on next page)

(continued from previous page)

```
    # learn more here: https://docs.ray.io/en/releases-0.7.1/tune-schedulers.
↪html
    scheduler = 'asha',

    # where to store logs
    log_dir = "RayLogs/",

    # resume hyper-parameter tuning from previous log
    # specify 'name' (i.e main_2021-03-09_18-33-56) below
    resume = False,

    # only required if you want to resume from previous checkpoint
    # it can also be specified if you don't want to resume
    name = None,

    # specify resources to be used per trial
    # i.e {'gpu':1, 'cpu':2}
    resources = {'gpu':0.5},

    # if True, trains model on Full dataset with the best parameter selected.
    final_train = True
    )
```

PYTHON MODULE INDEX

C

INDEX

D

E

F

G

L

M

N

O

R

S

T

U