DISTIL

Release v0.1

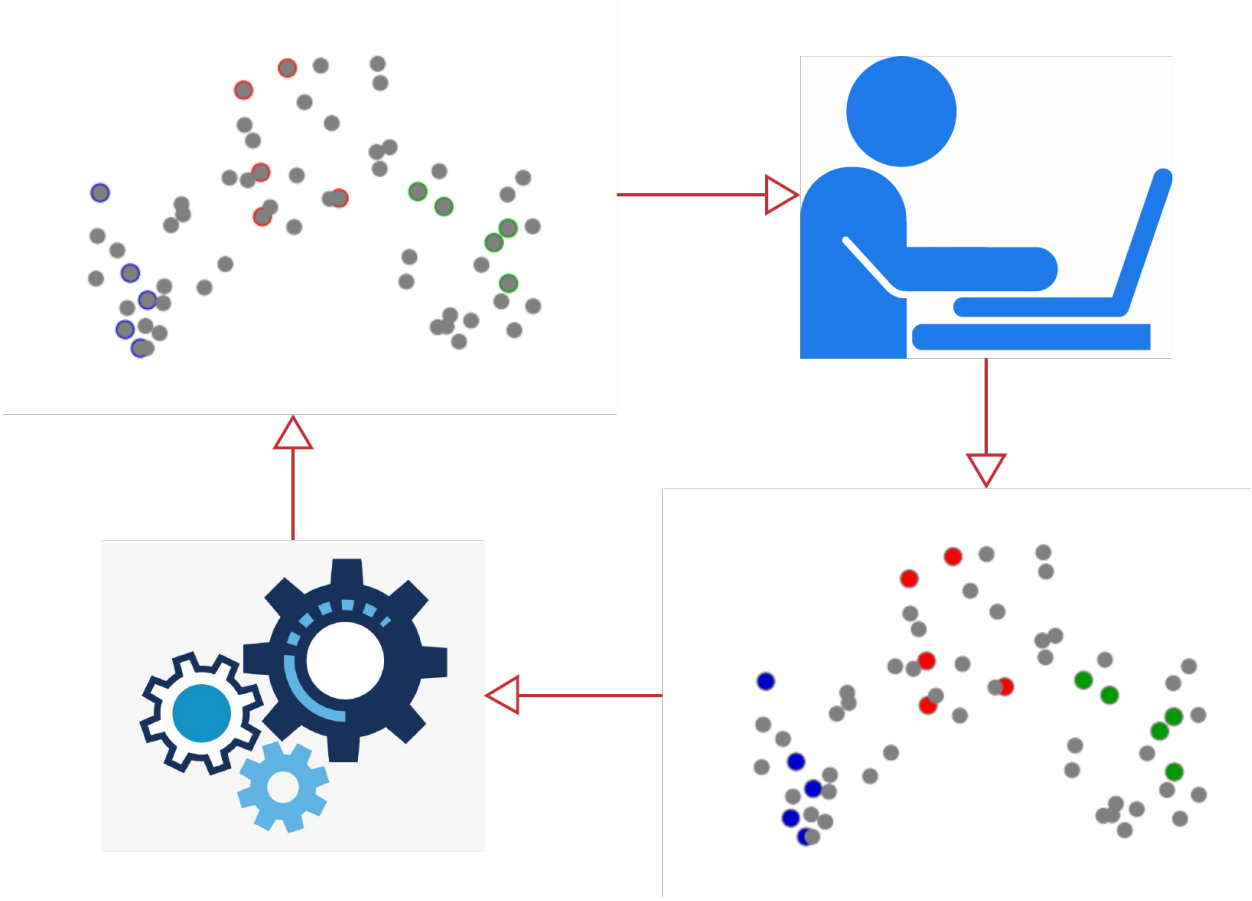
Durga Sivasubramanian,Nathan Beck,Apurva Dani, Rishabh Iyer

May 08, 2021

CONTENTS:

DISTIL:: Deep dIverSified inTeractIve Learning is an efficient and scalable active learning library built on top of PyTorch.

What is DISTIL?

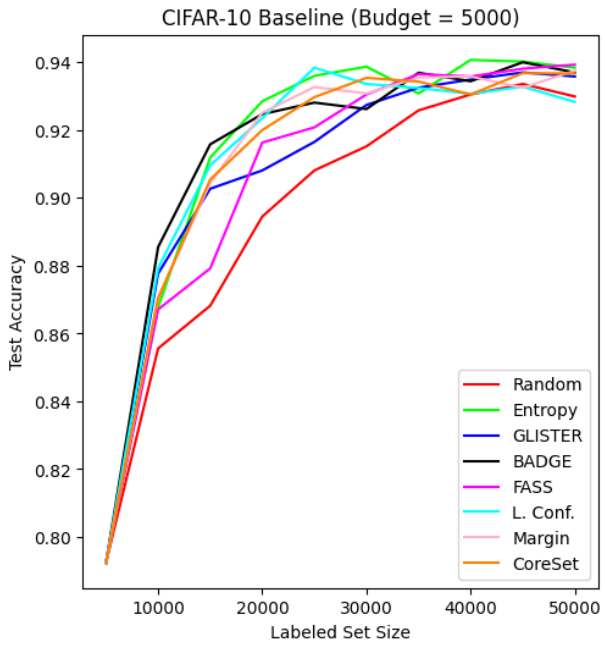


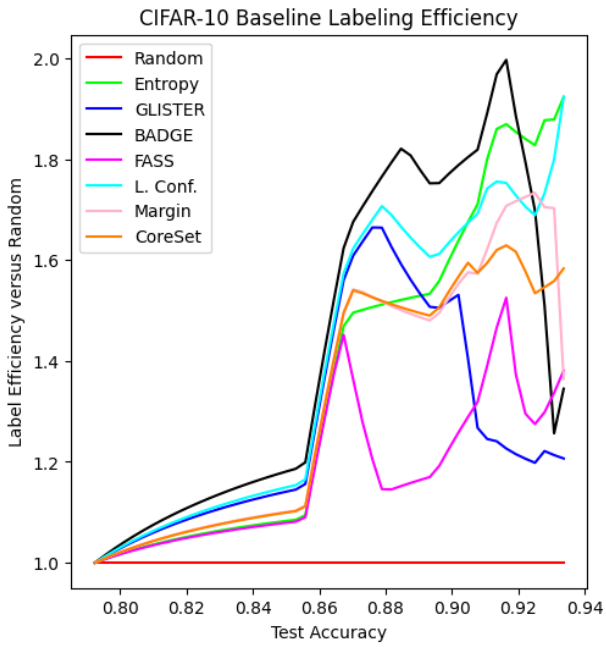
DISTIL is an active learning toolkit that implements a number of state-of-the-art active learning strategies with a particular focus for active learning in the deep learning setting. DISTIL is built on PyTorch and decouples the training loop from the active learning algorithm, thereby providing flexibility to the user by allowing them to control the training procedure and model. It allows users to incorporate new active learning algorithms easily with minimal changes to their existing code. DISTIL also provides support for incorporating active learning with your custom dataset and allows you to experiment on well-known datasets. We are continuously incorporating newer and better active learning selection strategies into DISTIL.

Principles of DISTIL:

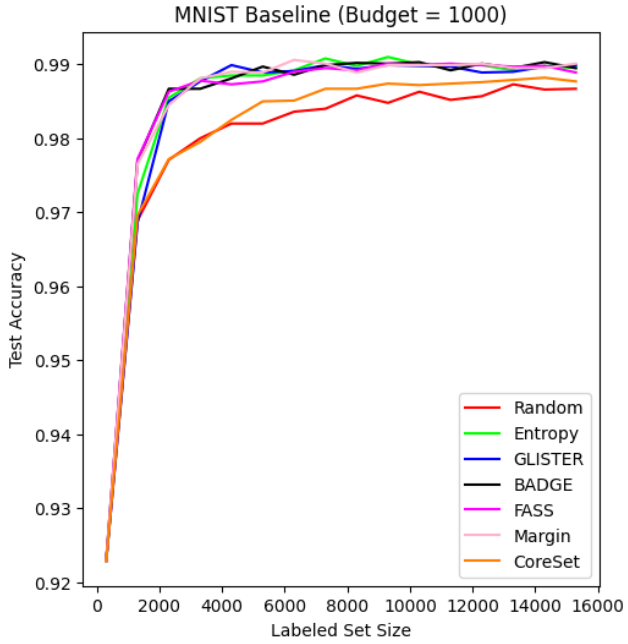
1. Minimal changes to add it to the existing training structure.
2. Independent of the training strategy used.
3. Achieving similar test accuracy with less amount of training data.
4. Huge reduction in labeling cost and time.
5. Access to various active learning strategies with just one line of code.

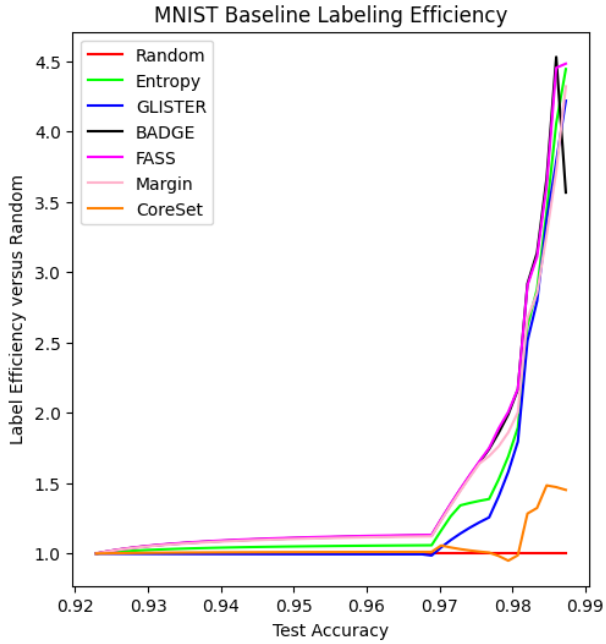
Preliminary Results: CIFAR-10



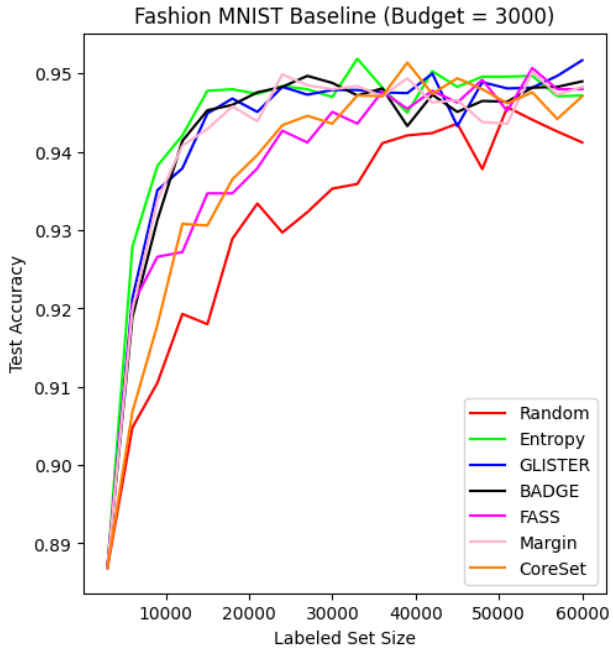


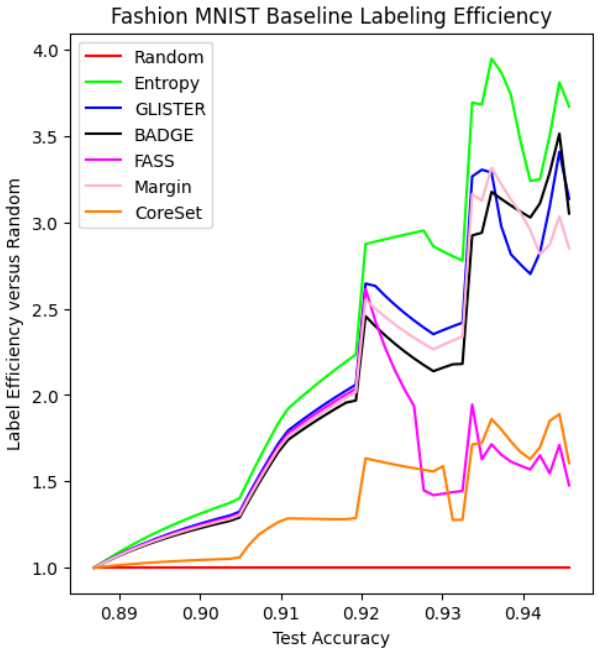
Preliminary Results: MNIST



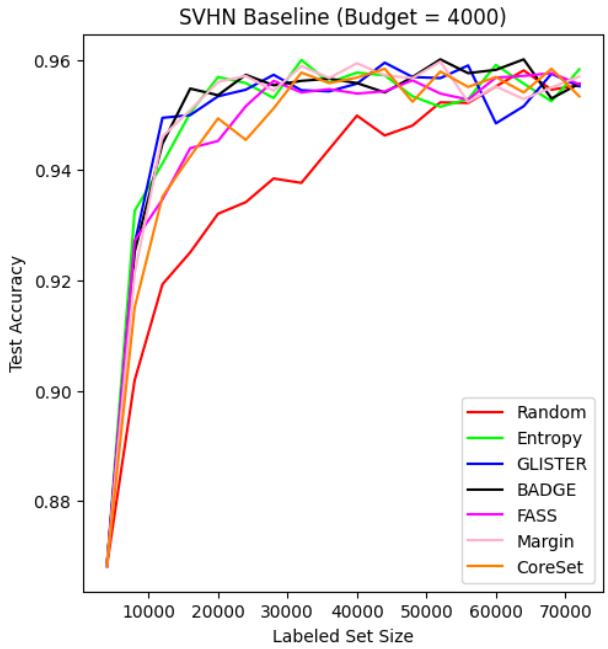


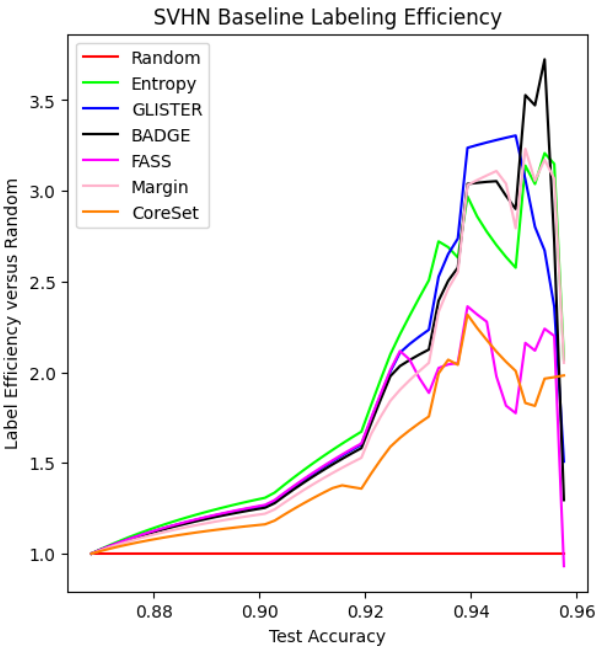
Preliminary Results: Fashion MNIST





Preliminary Results: SVHN





CHAPTER

ONE

DISTIL

1.1 Active Learning Strategies

1.1.1 BADGE

class distil.active_learning_strategies.badge.**BADGE**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args*)

Bases: distil.active_learning_strategies.strategy.Strategy

This method is based on the paper Deep Batch Active Learning by Diverse, Uncertain Gradient Lower Bounds[1]. According to the paper, this strategy, Batch Active learning by Diverse Gradient Embeddings (BADGE), samples groups of points that are disparate and high magnitude when represented in a hallucinated gradient space, a strategy designed to incorporate both predictive uncertainty and sample diversity into every selected batch. Crucially, BADGE trades off between uncertainty and diversity without requiring any hand-tuned hyperparameters. Here at each round of selection, loss gradients are computed using the hypothesised labels. Then to select the points to be labeled are selected by applying k-means++ on these loss gradients.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **Y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters. *batch_size* Batch size to be used inside strategy class (int, optional)

select(*budget*)

Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **chosen** – List of selected data point indexes with respect to unlabeled_x

Return type list

select_per_batch(*budget*, *batch_size*)

Select points to label by using per-batch BADGE strategy

[1] Jordan T. Ash, Chicheng Zhang, Akshay Krishnamurthy, John Langford, and Alekh Agarwal. Deep batch active learning by diverse, uncertain gradient lower bounds. *CoRR*, 2019. URL: http://arxiv.org/abs/1906.03671, arXiv:1906.03671.

Parameters

- **budget** (*int*) – Number of indices to be selected from unlabeled set
- **batch_size** (*int*) – Size of batches to form

Returns **chosen** – List of selected data point indices with respect to unlabeled_x

Return type list

1.1.2 Core-Set Approch

class distil.active_learning_strategies.core_set.**CoreSet**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args={}*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implementation of CoreSet[2] Strategy. A diversity-based approach using coreset selection. The embedding of each example is computed by the network's penultimate layer and the samples at each round are selected using a greedy furthest-first traversal conditioned on all labeled examples.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **Y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

furthest_first(*X*, *X_set*, *n*)

Selects points with maximum distance

Parameters

- **X** (*numpy array*) – Embeddings of unlabeled set
- **X_set** (*numpy array*) – Embeddings of labeled set
- **n** (*int*) – Number of points to return

Returns **idxs** – List of selected data point indexes with respect to unlabeled_x

Return type list

select(*budget*)

Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **chosen** – List of selected data point indexes with respect to unlabeled_x

Return type list

[2] Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: a core-set approach. 2018. arXiv:1708.00489.

1.1.3 CRAIG-ACTIVE

class distil.active_learning_strategies.craig_active.**CRAIGActive**(*X*, *Y*, *unlabeled_x*, *net*, *criterion*, *handler*, *nclasses*, *lrn_rate*, *selection_type*, *linear_layer*, *args={}*)

Bases: distil.active_learning_strategies.strategy.Strategy

This is an implementation of an active learning variant of CRAIG from the paper Coresets for Data-efficient Training of Machine Learning Models . This algorithm calculates hypothesized labels for each of the unlabeled points and feeds this hypothesized set to the original CRAIG algorithm. The selected points from CRAIG are used as the queried points for this algorithm.

Parameters

- **X** (*Numpy array*) – Features of the labled set of points
- **Y** (*Numpy array*) – Lables of the labled set of points
- **unlabeled_x** (*Numpy array*) – Features of the unlabled set of points
- **net** (*class object*) – Model architecture used for training. Could be instance of models defined in *distil.utils.models* or something similar.
- **criterion** (*class object*) – The loss type used in training. Could be instance of torch.nn.* losses or torch functionals.
- **handler** (*class object*) – It should be a subclass of torch.utils.data.Dataset i.e, have __getitem__ and __len__ methods implemented, so that is could be passed to pytorch DataLoader.Could be instance of handlers defined in *distil.utils.DataHandler* or something similar.
- **nclasses** (*int*) – No. of classes in tha dataset
- **lrn_rate** (*float*) – The learning rate used in training. Used by the CRAIG algorithm.
- **selection_type** (*string*) – Should be one of "PerClass", "Supervised", or "PerBatch". Selects which approximation method is used.
- **linear_layer** (*bool*) – Sets whether to include the last linear layer parameters as part of the gradient computation.
- **args** (*dictionary*) – This dictionary should have keys 'batch_size' and 'lr'. 'lr' should be the learning rate used for training. 'batch_size' 'batch_size' should be such that one can exploit the benefits of tensorization while honouring the resourse constraits.

select(*budget*)

Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **subset_idxs** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.4 Entropy Sampling

class distil.active_learning_strategies.entropy_sampling.**EntropySampling**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args={}*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implements the Entropy Sampling Strategy, one of the most basic active learning strategies, where we select samples about which the model is most uncertain. To quantify the uncertainity we use entropy and therefore select points which have maximum entropy.

Suppose the model has *nclasses* output nodes and each output node is denoted by z_j. Thus, $j \in [1, nclasses]$. Then for a output node z_i from the model, the corresponding softmax would be

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Then entropy can be calculated as,

$$ENTROPY = -\sum_j \sigma(z_j) * log(\sigma(z_i))$$

The algorithm then selects *budget* no. of elements with highest **ENTROPY**.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

select(*budget*)

Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **U_idx** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.5 Entropy Sampling with Dropout

class distil.active_learning_strategies.entropy_sampling_dropout.**EntropySamplingDropout**(*X*, *Y*, *unlabeled_net*, *handler*, *nclasses*, *args*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implements the Entropy Sampling Strategy with dropout. Entropy Sampling Strategy is one of the most basic active learning strategies, where we select samples about which the model is most uncertain. To quantify the uncertainity we use entropy and therefore select points which have maximum entropy.

Suppose the model has *nclasses* output nodes and each output node is denoted by z_j. Thus, $j \in [1, nclasses]$. Then for a output node z_i from the model, the corresponding softmax would be

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Then entropy can be calculated as,

$$ENTROPY = -\sum_j \sigma(z_j) * log(\sigma(z_i))$$

The algorithm then selects *budget* no. of elements with highest **ENTROPY**.

The drop out version uses the predict probability dropout function from the base strategy class to find the hypothesised labels. User can pass n_drop argument which denotes the number of times the probabilities will be calculated. The final probability is calculated by averaging probabilities obtained in all iteraitons.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

  n_drop Dropout value to be used (int, optional)

select(*budget*)

Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **U_idx** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.6 FASS

class distil.active_learning_strategies.fass.**FASS**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args={}*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implements FASS[3] combines the uncertainty sampling method with a submodular data subset selection framework to label a subset of data points to train a classifier. Here the based on the 'top_n' parameter, 'top_n*budget' most uncertain parameters are filtered. On these filtered points one of the submodular functions viz. 'facility_location' , 'graph_cut', 'saturated_coverage', 'sum_redundancy', 'feature_based' is applied to get the final set of points.

We select a subset F of size β based on uncertainty sampling, such that $\beta \geq k$.

Then select a subset S by solving

$$\max\{f(S) \text{ such that } |S| \leq k, S \subseteq F\}$$

where k is the is the *budget* and f can be one of these functions - 'facility location' , 'graph cut', 'saturated coverage', 'sum redundancy', 'feature based'.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters - *batch_size* Batch size to be used inside strategy class (int, optional)
- **submod** (*str*) –
- **of submodular function - 'facility_location' | 'graph_cut' | 'saturated_coverage' | 'sum_redundancy' | 'feature_based'** (*Choice*) –
- **selection_type** (*str*) –
- **of selection strategy - 'PerClass' | 'Supervised'** (*Choice*) –

select(*budget*, *top_n=5*)

Select next set of points

Parameters

- **budget** (*int*) – Number of indexes to be returned for next set
- **top_n** (*float*) – It is the multiper to the budget which decides the size of the data points on which submodular functions will be applied. For example top_n = 5, if 5*budget points will be passed to the submodular functions.

Returns **return_indices** – List of selected data point indexes with respect to unlabeled_x

Return type list

[3] Kai Wei, Rishabh Iyer, and Jeff Bilmes. Submodularity in data subset selection and active learning. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of Proceedings of Machine Learning Research, 1954–1963. Lille, France, 07–09 Jul 2015. PMLR. URL: http://proceedings.mlr.press/v37/wei15.html.

1.1.7 GLISTER

class distil.active_learning_strategies.glister.**GLISTER**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args*, *valid*, *X_val=None*, *Y_val=None*, *loss_criterion=CrossEntropyLoss()*, *typeOf='none'*, *lam=None*, *kernel_batch_size=200*)

Bases: distil.active_learning_strategies.strategy.Strategy

This is implementation of GLISTER-ACTIVE from the paper GLISTER: Generalization based Data Subset Selection for Efficient and Robust Learning[4]. GLISTER methods tries to solve a bi-level optimisation problem.

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

In the above equation, $\mathcal{U}$ denotes the Data without lables i.e. *unlabeled_x*, $\mathcal{V}$ 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*. 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.

Parameters

- **X** (*Numpy array*) – Features of the labled set of points
- **Y** (*Numpy array*) – Lables of the labled set of points
- **unlabeled_x** (*Numpy array*) – Features of the unlabled set of points
- **net** (*class object*) – Model architecture used for training. Could be instance of models defined in *distil.utils.models* or something similar.
- **handler** (*class object*) – It should be a subclass of torch.utils.data.Dataset i.e, have __getitem__ and __len__ methods implemented, so that is could be passed to pytorch DataLoader.Could be instance of handlers defined in *distil.utils.DataHandler* or something similar.
- **nclasses** (*int*) – No. of classes in tha dataset
- **args** (*dictionary*) – This dictionary should have keys 'batch_size' and 'lr'. 'lr' should be the learning rate used for training. 'batch_size' 'batch_size' should be such that one can exploit the benefits of tensorization while honouring the resourse constraits.
- **valid** (*boolean*) – Whether validation set is passed or not
- **X_val** (*Numpy array, optional*) – Features of the points in the validation set. Mandatory if *valid=True*.

[4] Krishnateja Killamsetty, Durga Sivasubramanian, Ganesh Ramakrishnan, and Rishabh Iyer. Glister: generalization based data subset selection for efficient and robust learning. 2020. arXiv:2012.10630.

- **Y_val** (*Numpy array, optional*) – Lables of the points in the validation set. Mandatory if *valid=True*.
- **loss_criterion** (*class object, optional*) – The type of loss criterion. Default is **torch.nn.CrossEntropyLoss()**
- **typeOf** (*str, optional*) – Determines the type of regulariser to be used. Default is **'none'**. For random regulariser use **'Rand'**. To use Facility Location set functiom as a regulariser use **'FacLoc'**. To use Diversity set functiom as a regulariser use **'Diversity'**.
- **lam** (*float, optional*) – Determines the amount of regularisation to be applied. Mandatory if is not *typeOf='none'* and by default set to *None*. For random regulariser use values should be between 0 and 1 as it determines fraction of points replaced by random points. For both 'Diversity' and 'FacLoc', *lam* determines the weightage given to them while computing the gain.
- **kernel_batch_size** (*int, optional*) – For 'Diversity' and 'FacLoc' regualrizer versions, similarity kernel is to be computed, which entails creating a 3d torch tensor of dimenssions kernel_batch_size*kernel_batch_size* feature dimenssion.Again kernel_batch_size should be such that one can exploit the benefits of tensorization while honouring the resourse constraits.

select (*budget*)

Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **chosen** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.8 GRADMATCH

class distil.active_learning_strategies.gradmatch_active.**GradMatchActive**(*X*, *Y*, *unlabeled_x*, *net*, *criterion*, *handler*, *nclasses*, *lrn_rate*, *selection_type*, *linear_layer*, *args={}*, *valid=False*, *X_val=None*, *Y_val=None*)

Bases: distil.active_learning_strategies.strategy.Strategy

This is an implementation of an active learning variant of GradMatch from the paper GRAD-MATCH: A Gradient Matching Based Data Subset Selection for Efficient Learning . This algorithm solves a fixed-weight version of the error term present in the paper by a greedy selection algorithm akin to the original GradMatch's Orthogonal Matching Pursuit. The gradients computed are on the hypothesized labels of the loss function and are matched to either the full gradient of these hypothesized examples or a supplied validation gradient. The indices returned are the ones selected by this algorithm.

$$Err(X_t, L, L_T, \theta_t) = \left|\left| \sum_{i \in X_t} \nabla_\theta L_T^i(\theta_t) - \frac{k}{N} \nabla_\theta L(\theta_t) \right|\right|$$

where,

- Each gradient is computed with respect to the last layer's parameters
- θ_t are the model parameters at selection round t
- X_t is the queried set of points to label at selection round t
- k is the budget
- N is the number of points contributing to the full gradient $\nabla_\theta L(\theta_t)$
- $\nabla_\theta L(\theta_t)$ is either the complete hypothesized gradient or a validation gradient
- $\sum_{i \in X_t} \nabla_\theta L_T^i(\theta_t)$ is the subset's hypothesized gradient with $|X_t| = k$

Parameters

- **X** (*Numpy array*) – Features of the labled set of points
- **Y** (*Numpy array*) – Lables of the labled set of points
- **unlabeled_x** (*Numpy array*) – Features of the unlabled set of points
- **net** (*class object*) – Model architecture used for training. Could be instance of models defined in *distil.utils.models* or something similar.
- **criterion** (*class object*) – The loss type used in training. Could be instance of torch.nn.* losses or torch functionals.
- **handler** (*class object*) – It should be a subclass of torch.utils.data.Dataset i.e, have __getitem__ and __len__ methods implemented, so that is could be passed to pytorch DataLoader.Could be instance of handlers defined in *distil.utils.DataHandler* or something similar.
- **nclasses** (*int*) – No. of classes in tha dataset
- **lrn_rate** (*float*) – The learning rate used in training. Used by the original GradMatch algorithm.
- **selection_type** (*string*) – Should be one of "PerClass" or "PerBatch". Selects which approximation method is used.
- **linear_layer** (*bool*) – Sets whether to include the last linear layer parameters as part of the gradient computation.
- **args** (*dictionary*) – This dictionary should have keys 'batch_size' and 'lr'. 'lr' should be the learning rate used for training. 'batch_size' 'batch_size' should be such that one can exploit the benefits of tensorization while honouring the resourse constraits.
- **valid** (*boolean*) – Whether validation set is passed or not
- **X_val** (*Numpy array, optional*) – Features of the points in the validation set. Mandatory if *valid=True*.

- **Y_val** (*Numpy array, optional*) – Lables of the points in the validation set. Mandatory if *valid=True*.

select (*budget*, *use_weights*)
Select next set of points

Parameters

- **budget** (*int*) – Number of indexes to be returned for next set
- **use_weights** (*bool*) – Whether to use fixed-weight version (false) or OMP version (true)

Returns **subset_idxs** – List of selected data point indexes with respect to unlabeled_x and, if use_weights is true, the weights associated with each point

Return type list

1.1.9 Least Confidence

class distil.active_learning_strategies.least_confidence.**LeastConfidence**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args={}*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implements the Least Confidence Sampling Strategy a active learning strategy where the algorithm selects the data points for which the model has the lowest confidence while predicting its label.

Suppose the model has *nclasses* output nodes denoted by $\overrightarrow{z}$ and each output node is denoted by z_j. Thus, $j \in [1, nclasses]$. Then for a output node z_i from the model, the corresponding softmax would be

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Then the softmax can be used pick *budget* no. of elements for which the model has the lowest confidence as follows,

$$\text{argmin}_{S \subseteq \mathcal{U}, |S| \leq k} \sum_{S} (\text{argmax}_j(\sigma(\overrightarrow{z})))$$

where $\mathcal{U}$ denotes the Data without lables i.e. *unlabeled_x* and k is the *budget*.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables

- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

select (*budget*)

Select next set of points

Parameters **budget** (*int*) – Nuber of indexes to be returned for next set

Returns **U_idx** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.10 Least Confidence with Dropout

class distil.active_learning_strategies.least_confidence_dropout.**LeastConfidenceDropout**(*X*, *Y*, *unlabeled*, *net*, *handler*, *nclasses*, *args*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implements the Least Confidence Sampling Strategy with dropout a active learning strategy where the algorithm selects the data points for which the model has the lowest confidence while predicting its label.

Suppose the model has *nclasses* output nodes denoted by $\overrightarrow{z}$ and each output node is denoted by z_j. Thus, $j \in [1, nclasses]$. Then for a output node z_i from the model, the corresponding softmax would be

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Then the softmax can be used pick *budget* no. of elements for which the model has the lowest confidence as follows,

$$\text{argmin}_{S \subseteq \mathcal{U}, |S| \leq k} \sum_S (\text{argmax}_j (\sigma(\overrightarrow{z})))$$

where $\mathcal{U}$ denotes the Data without lables i.e. *unlabeled_x* and k is the *budget*.

The drop out version uses the predict probability dropout function from the base strategy class to find the hypothesised labels. User can pass n_drop argument which denotes the number of times the probabilities will be calculated. The final probability is calculated by averaging probabilities obtained in all iteraitons.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables

- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

  n_drop Dropout value to be used (int, optional)

select (*budget*)

Select next set of points

Parameters **budget** (*int*) – Nuber of indexes to be returned for next set

Returns **U_idx** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.11 Margin Sampling

class distil.active_learning_strategies.margin_sampling.**MarginSampling**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args={}*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implements the Margin Sampling Strategy a active learning strategy similar to Least Confidence Sampling Strategy. While least confidence only takes into consideration the maximum probability, margin sampling considers the difference between the confidence of first and the second most probable labels.

Suppose the model has *nclasses* output nodes denoted by $\overrightarrow{z}$ and each output node is denoted by z_j. Thus, $j \in [1, nclasses]$. Then for a output node z_i from the model, the corresponding softmax would be

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Let,

$$m = \text{argmax}_j(\sigma(\overrightarrow{z}))$$

Then using softmax, Margin Sampling Strategy would pick *budget* no. of elements as follows,

$$\text{argmin}_{S \subseteq \mathcal{U}, |S| \leq k} \sum_S (\text{argmax}_j(\sigma(\overrightarrow{z}))) - (\text{argmax}_{j \neq m}(\sigma(\overrightarrow{z})))$$

where $\mathcal{U}$ denotes the Data without lables i.e. *unlabeled_x* and k is the *budget*.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.

- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

select (*budget*)

Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **U_idx** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.12 Margin sampling with Dropout

class distil.active_learning_strategies.margin_sampling_dropout.**MarginSamplingDropout**(*X*, *Y*, *unlabeled_*, *net*, *handler*, *nclass*, *args=*{

Bases: distil.active_learning_strategies.strategy.Strategy

Implements the Margin Sampling Strategy with dropout a active learning strategy similar to Least Confidence Sampling Strategy with dropout. While least confidence only takes into consideration the maximum probability, margin sampling considers the difference between the confidence of first and the second most probable labels.

Suppose the model has *nclasses* output nodes denoted by $\overrightarrow{z}$ and each output node is denoted by z_j. Thus, $j \in [1, nclasses]$. Then for a output node z_i from the model, the corresponding softmax would be

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Let,

$$m = \text{argmax}_j(\sigma(\overrightarrow{z}))$$

Then using softmax, Margin Sampling Strategy would pick *budget* no. of elements as follows,

$$\text{argmin}_{S \subseteq \mathcal{U}, |S| \leq k} \sum_S (\text{argmax}_j(\sigma(\overrightarrow{z}))) - (\text{argmax}_{j \neq m}(\sigma(\overrightarrow{z})))$$

where $\mathcal{U}$ denotes the Data without lables i.e. *unlabeled_x* and k is the *budget*.

The drop out version uses the predict probability dropout function from the base strategy class to find the hypothesised labels. User can pass n_drop argument which denotes the number of times the probabilities will be calculated. The final probability is calculated by averaging probabilities obtained in all iteraitons.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels

- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

  n_drop Dropout value to be used (int, optional)

select (*budget*)

Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **U_idx** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.13 Random Sampling

class distil.active_learning_strategies.random_sampling.**RandomSampling**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args={}*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implementation of Random Sampling Strategy. This strategy is often used as a baseline, where we pick a set of unlabeled points randomly.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

select (*budget*)

Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **rand_idx** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.14 Submodular Sampling

class distil.active_learning_strategies.submod_sampling.**SubmodSampling**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *typeOf*, *selection_type*, *if_grad=False*, *args={}*, *kernel_batch_size=200*)

Bases: distil.active_learning_strategies.strategy.Strategy

This strategy uses one of the submodular functions viz. 'facility_location', 'graph_cut', 'saturated_coverage', 'sum_redundancy', 'feature_based'[5] or Disparity-sum, Disparity-min[6] or DPP[7] is used to select the points to be labeled. These techniques can be applied directly to the features/embeddings or on the gradients of the loss functions.

Parameters

- **X** (*Numpy array*) – Features of the labled set of points
- **Y** (*Numpy array*) – Lables of the labled set of points
- **unlabeled_x** (*Numpy array*) – Features of the unlabled set of points
- **net** (*class object*) – Model architecture used for training. Could be instance of models defined in *distil.utils.models* or something similar.
- **handler** (*class object*) – It should be a subclass of torch.utils.data.Dataset i.e, have __getitem__ and __len__ methods implemented, so that is could be passed to pytorch DataLoader.Could be instance of handlers defined in *distil.utils.DataHandler* or something similar.
- **nclasses** (*int*) – No. of classes in tha dataset
- **typeOf** (*str*) – Choice of submodular function - 'facility_location' | 'graph_cut' | 'saturated_coverage' | 'sum_redundancy' | 'feature_based' | 'Disparity-min' | 'Disparity-sum' | 'DPP'
- **selection_type** (*str*) – selection strategy - 'Full' |'PerClass' | 'Supervised'
- **if_grad** (*boolean, optional*) – Determines if gradients to be used for subset selection. Default is False.

[5] Rishabh Iyer, Ninad Khargonkar, Jeff Bilmes, and Himanshu Asnani. Submodular combinatorial information measures with applications in machine learning. 2021. arXiv:2006.15412.

[6] Anirban Dasgupta, Ravi Kumar, and Sujith Ravi. Summarization through submodularity and dispersion. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 1014–1022. Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/P13-1100.

[7] Laming Chen, Guoxin Zhang, and Eric Zhou. Fast greedy map inference for determinantal point process to improve recommendation diversity. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/dbbf603ff0e99629dda5d75b6f75f966-Paper.pdf.

- **args** (*dictionary*) – This dictionary should have keys 'batch_size' and 'lr'. 'lr' should be the learning rate used for training. 'batch_size' 'batch_size' should be such that one
- **kernel_batch_size** (*int, optional*) – For 'Diversity' and 'FacLoc' regualrizer versions, similarity kernel is to be computed, which entails creating a 3d torch tensor of dimenssions kernel_batch_size*kernel_batch_size* feature dimenssion.Again kernel_batch_size should be such that one can exploit the benefits of tensorization while honouring the resourse constraits.

select (*budget*)
Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **chosen** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.15 Adversarial BIM

class distil.active_learning_strategies.adversarial_bim.**AdversarialBIM**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args={}*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implements Adversial Bim Strategy which is motivated by the fact that often the distance computation from decision boundary is difficult and intractable for margin based methods. This technique avoids estimating distance by using BIM(Basic Iterative Method)[8] to estimate how much adversarial perturbation is required to cross the boundary. Smaller the required the perturbation, closer the point is to the boundary.

Basic Iterative Method (BIM): Given a base input, the approach is to perturb each feature in the direction of the gradient by magnitude ϵ, where is a parameter that determines perturbation size. For a model with loss $\nabla J(\theta, x, y)$, where θ represents the model parameters, x is the model input, and y is the label of x, the adversarial sample is generated iteratively as,

$$\begin{aligned} & to \\ x_0^* &= x, \\ x_i^* &= clip_{x,e}(x_{i-1}^* + sign(\nabla_{x_{i-1}^*} J(\theta, x_{i-1}^*, y))) \end{aligned} \tag{1.2}$$

Parameters

- **X** (*numpy array*) – Present training/labeled data

[8] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters

  batch_size- Batch size to be used inside strategy class (int, optional)

  eps-epsilon value for gradients

select(*budget*)

Selects next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **idxs** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.16 Adversarial DeepFool

class distil.active_learning_strategies.adversarial_deepfool.**AdversarialDeepFool**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args={}*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implements Adversial Deep Fool Strategy[9], a Deep-Fool based Active Learning strategy that selects unlabeled samples with the smallest adversarial perturbation. This technique is motivated by the fact that often the distance computation from decision boundary is difficult and intractable for margin-based methods. This technique avoids estimating distance by using Deep-Fool[10] like techniques to estimate how much adversarial perturbation is required to cross the boundary. The smaller the required perturbation, the closer the point is to the boundary.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables

[9] Melanie Ducoffe and Frederic Precioso. Adversarial active learning for deep networks: a margin based approach. 2018. arXiv:1802.09841.

[10] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

  max_iter Maximum Number of Iterations (int, optional)

select (*budget*)
Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **idxs** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.17 Bayesian Active Learning Disagreement Dropout

class distil.active_learning_strategies.bayesian_active_learning_disagreement_dropout.**BALDI**

Bases: distil.active_learning_strategies.strategy.Strategy

Implements Bayesian Active Learning by Disagreement (BALD) Strategy[11], which assumes a Basiyan setting and selects points which maximise the mutual information between the predicted labels and model parameters. This implementation is an adaptation for a non-bayesian setting, with the assumption that there is a dropout layer in the model.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

  n_drop Dropout value to be used (int, optional)

select (*budget*)
Select next set of points

Parameters **budget** (*int*) – Number of indexes to be returned for next set

Returns **idxs** – List of selected data point indexes with respect to unlabeled_x

[11] Neil Houlsby, Ferenc Huszár, Zoubin Ghahramani, and Máté Lengyel. Bayesian active learning for classification and preference learning. 2011. arXiv:1112.5745.

Return type list

1.1.18 KMeans Sampling

class distil.active_learning_strategies.kmeans_sampling.**KMeansSampling**(*X*, *Y*, *unlabeled_x*, *net*, *handler*, *nclasses*, *args={}*)

Bases: distil.active_learning_strategies.strategy.Strategy

Implements KMeans Sampling selection strategy, the last layer embeddings are calculated for all the unlabeled data points.Then the KMeans clustering algorithm is run over these embeddings with the number of clusters equal to the *budget*. Then the distance is calculated for all the points from their respective centers. From each cluster, the point closest to the center is selected to be labeled for the next iteration. Since the number of centers are equal to the budget, selecting one point from each cluster satisfies the total number of data points to be selected in one iteration.

Parameters

- **X** (*numpy array*) – Present training/labeled data
- **y** (*numpy array*) – Labels of present training data
- **unlabeled_x** (*numpy array*) – Data without labels
- **net** (*class*) – Pytorch Model class
- **handler** (*class*) – Data Handler, which can load data even without labels.
- **nclasses** (*int*) – Number of unique target variables
- **args** (*dict*) – Specify optional parameters

  batch_size Batch size to be used inside strategy class (int, optional)

select(*budget*)

Select next set of points

Parameters **budget** (*int*) – Nuber of indexes to be returned for next set

Returns **q_idxs** – List of selected data point indexes with respect to unlabeled_x

Return type list

1.1.19 REFERENCES

1.2 utils

1.2.1 DataHandler

class distil.utils.data_handler.**DataHandler_CIFAR10**(*X*, *Y=None*, *select=True*, *use_test_transform=False*)

Bases: Generic[torch.utils.data.dataset.T_co]

Data Handler to load CIFAR10 dataset. This class extends `torch.utils.data.Dataset` to handle loading data even without labels

Parameters

- **X** (*numpy array*) – Data to be loaded
- **y** (*numpy array, optional*) – Labels to be loaded (default: None)
- **select** (*bool*) – True if loading data without labels, False otherwise
- **use_test_transform** (*bool*) – True if the data handler should apply the test transform. Otherwise, the data handler will use the training transform (default: False)

class distil.utils.data_handler.**DataHandler_CIFAR100**(*X*, *Y=None*, *select=True*, *use_test_transform=False*)

Bases: Generic[torch.utils.data.dataset.T_co]

Data Handler to load CIFAR100 dataset. This class extends `torch.utils.data.Dataset` to handle loading data even without labels

Parameters

- **X** (*numpy array*) – Data to be loaded
- **y** (*numpy array, optional*) – Labels to be loaded (default: None)
- **select** (*bool*) – True if loading data without labels, False otherwise
- **use_test_transform** (*bool*) – True if the data handler should apply the test transform. Otherwise, the data handler will use the training transform (default: False)

class distil.utils.data_handler.**DataHandler_FASHION_MNIST**(*X*, *Y=None*, *select=True*, *use_test_transform=False*)

Bases: Generic[torch.utils.data.dataset.T_co]

Data Handler to load FASHION_MNIST dataset. This class extends `torch.utils.data.Dataset` to handle loading data even without labels

Parameters

- **X** (*numpy array*) – Data to be loaded
- **y** (*numpy array, optional*) – Labels to be loaded (default: None)
- **select** (*bool*) – True if loading data without labels, False otherwise
- **use_test_transform** (*bool*) – True if the data handler should apply the test transform. Otherwise, the data handler will use the training transform (default: False)

class distil.utils.data_handler.**DataHandler_KMNIST**(*X*, *Y=None*, *select=True*, *use_test_transform=False*)

Bases: Generic[torch.utils.data.dataset.T_co]

Data Handler to load KMNIST dataset. This class extends `torch.utils.data.Dataset` to handle loading data even without labels

Parameters

- **X** (*numpy array*) – Data to be loaded
- **y** (*numpy array, optional*) – Labels to be loaded (default: None)
- **select** (*bool*) – True if loading data without labels, False otherwise
- **use_test_transform** (*bool*) – True if the data handler should apply the test transform. Otherwise, the data handler will use the training transform (default: False)

class distil.utils.data_handler.**DataHandler_MNIST**(*X*, *Y=None*, *select=True*, *use_test_transform=False*)

Bases: Generic[torch.utils.data.dataset.T_co]

Data Handler to load MNIST dataset. This class extends `torch.utils.data.Dataset` to handle loading data even without labels

Parameters

- **X** (*numpy array*) – Data to be loaded
- **y** (*numpy array, optional*) – Labels to be loaded (default: None)
- **select** (*bool*) – True if loading data without labels, False otherwise
- **use_test_transform** (*bool*) – True if the data handler should apply the test transform. Otherwise, the data handler will use the training transform (default: False)

class distil.utils.data_handler.**DataHandler_Points**(*X*, *Y=None*, *select=True*, *use_test_transform=False*)

Bases: Generic[torch.utils.data.dataset.T_co]

Data Handler to load data points. This class extends `torch.utils.data.Dataset` to handle loading data even without labels

Parameters

- **X** (*numpy array*) – Data to be loaded
- **y** (*numpy array, optional*) – Labels to be loaded (default: None)
- **select** (*bool*) – True if loading data without labels, False otherwise

class distil.utils.data_handler.**DataHandler_STL10**(*X*, *Y=None*, *select=True*, *use_test_transform=False*)

Bases: Generic[torch.utils.data.dataset.T_co]

Data Handler to load STL10 dataset. This class extends `torch.utils.data.Dataset` to handle loading data even without labels

Parameters

- **X** (*numpy array*) – Data to be loaded
- **y** (*numpy array, optional*) – Labels to be loaded (default: None)
- **select** (*bool*) – True if loading data without labels, False otherwise
- **use_test_transform** (*bool*) – True if the data handler should apply the test transform. Otherwise, the data handler will use the training transform (default: False)

class distil.utils.data_handler.**DataHandler_SVHN**(*X*, *Y=None*, *select=True*, *use_test_transform=False*)

Bases: Generic[torch.utils.data.dataset.T_co]

Data Handler to load SVHN dataset. This class extends `torch.utils.data.Dataset` to handle loading data even without labels

Parameters

- **X** (*numpy array*) – Data to be loaded
- **y** (*numpy array, optional*) – Labels to be loaded (default: None)
- **select** (*bool*) – True if loading data without labels, False otherwise
- **use_test_transform** (*bool*) – True if the data handler should apply the test transform. Otherwise, the data handler will use the training transform (default: False)

1.2.2 Dataset

distil.utils.dataset.**add_label_noise**(*y_trn*, *num_cls*, *noise_ratio=0.8*)

Adds noise to the specified list of labels. This functionality is taken from CORDS and applied here.

Parameters

- **y_trn** (*list*) – The list of labels to add noise.
- **num_cls** (*int*) – The number of classes possible in the list.
- **noise_ratio** (*float, optional*) – The percentage of labels to modify. The default is 0.8.

Returns **y_trn** – The list of now-noisy labels

Return type list

distil.utils.dataset.**get_CIFAR10**(*path*, *tr_load_args=None*, *te_load_args=None*)

Downloads CIFAR10 dataset

Parameters **path** (*str*) – Path to save the downloaded dataset

Returns

- **X_tr** (*numpy array*) – Train set
- **Y_tr** (*torch tensor*) – Training Labels
- **X_te** (*numpy array*) – Test Set
- **Y_te** (*torch tensor*) – Test labels

distil.utils.dataset.**get_CIFAR100**(*path*, *tr_load_args=None*, *te_load_args=None*)

Downloads CIFAR100 dataset

Parameters **path** (*str*) – Path to save the downloaded dataset

Returns

- **X_tr** (*numpy array*) – Train set
- **Y_tr** (*torch tensor*) – Training Labels
- **X_te** (*numpy array*) – Test Set
- **Y_te** (*torch tensor*) – Test labels

distil.utils.dataset.**get_FASHION_MNIST**(*path*, *tr_load_args=None*, *te_load_args=None*)
Downloads FASHION_MNIST dataset

Parameters **path** (*str*) – Path to save the downloaded dataset

Returns

- **X_tr** (*numpy array*) – Train set
- **Y_tr** (*torch tensor*) – Training Labels
- **X_te** (*numpy array*) – Test Set
- **Y_te** (*torch tensor*) – Test labels

distil.utils.dataset.**get_KMNIST**(*path*, *tr_load_args=None*, *te_load_args=None*)
Downloads KMNIST dataset

Parameters **path** (*str*) – Path to save the downloaded dataset

Returns

- **X_tr** (*numpy array*) – Train set
- **Y_tr** (*torch tensor*) – Training Labels
- **X_te** (*numpy array*) – Test Set
- **Y_te** (*torch tensor*) – Test labels

distil.utils.dataset.**get_MNIST**(*path*, *tr_load_args=None*, *te_load_args=None*)
Downloads MNIST dataset

Parameters **path** (*str*) – Path to save the downloaded dataset

Returns

- **X_tr** (*numpy array*) – Train set
- **Y_tr** (*torch tensor*) – Training Labels
- **X_te** (*numpy array*) – Test Set
- **Y_te** (*torch tensor*) – Test labels

distil.utils.dataset.**get_STL10**(*path*, *tr_load_args=None*, *te_load_args=None*)
Downloads STL10 dataset

Parameters **path** (*str*) – Path to save the downloaded dataset

Returns

- **X_tr** (*numpy array*) – Train set
- **Y_tr** (*torch tensor*) – Training Labels
- **X_te** (*numpy array*) – Test Set
- **Y_te** (*torch tensor*) – Test labels

distil.utils.dataset.**get_SVHN**(*path*, *tr_load_args=None*, *te_load_args=None*)
Downloads SVHN dataset

Parameters **path** (*str*) – Path to save the downloaded dataset

Returns

- **X_tr** (*numpy array*) – Train set
- **Y_tr** (*torch tensor*) – Training Labels

- **X_te** (*numpy array*) – Test Set
- **Y_te** (*torch tensor*) – Test labels

distil.utils.dataset.**get_dataset**(*name*, *path*, *tr_load_args=None*, *te_load_args=None*)
Loads dataset

Parameters

- **name** (*str*) – Name of the dataset to be loaded. Supports MNIST and CIFAR10
- **path** (*str*) – Path to save the downloaded dataset
- **tr_load_args** (*dict*) – String dictionary for train distribution shift loading
- **te_load_args** (*dict*) – String dictionary for test distribution shift loading

Returns

- **X_tr** (*numpy array*) – Train set
- **Y_tr** (*torch tensor*) – Training Labels
- **X_te** (*numpy array*) – Test Set
- **Y_te** (*torch tensor*) – Test labels

distil.utils.dataset.**get_imbalanced_idx**(*y_trn*, *num_cls*, *class_ratio=0.6*)
Returns a list of indices of the supplied dataset that constitute a class-imbalanced subset of the supplied dataset. This functionality is taken from CORDS and applied here.

Parameters

- **y_trn** (*numpy ndarray*) – The label set to choose imbalance.
- **num_cls** (*int*) – The number of classes possible in the list.
- **class_ratio** (*float, optional*) – The percentage of classes to affect. The default is 0.6.

Returns **subset_idxs** – The list of indices of the supplied dataset that constitute a class-imbalanced subset

Return type list

distil.utils.dataset.**make_data_redundant**(*X*, *Y*, *intial_bud*, *unique_points=5000*, *amtRed=2*)
Modifies the input dataset in such a way that only X.shape(0)/amtRed are original points and rest are repeated or redundant.

Parameters

- **X** (*numpy ndarray*) – The feature set to be made redundant.
- **Y** (*numpy ndarray*) – The label set corresponding to the X.
- **intial_bud** (*int*) – Number of inital points that are assumed to be labled.
- **unique_points** (*int*) – Number of points to be kept unique in unlabled pool.
- **amtRed** (*float, optional*) – Factor that determines redundancy. The default is 2.

Returns **X** – Modified feature set.

Return type numpy ndarray

1.2.3 Submodular Functions

class distil.utils.submodular.**SubmodularFunction**(*device*, *x_trn*, *y_trn*, *N_trn*, *batch_size*, *submod*, *selection_type*)

Bases: *distil.utils.similarity_mat.SimilarityComputation*

Implementation of Submodular Function. This class allows you to use different submodular functions

Parameters

- **device** (*str*) – Device to be used, cpu|gpu
- **x_trn** (*torch tensor*) – Data on which submodular optimization should be applied
- **y_trn** (*torch tensor*) – Labels of the data
- **model** (*class*) – Model architecture used for training
- **N_trn** (*int*) – Number of samples in dataset
- **batch_size** (*int*) – Batch size to be used for optimization
- **if_convex** (*bool*) – If convex or not
- **submod** (*str*) – Choice of submodular function - 'facility_location' | 'graph_cut' | 'saturated_coverage' | 'sum_redundancy' | 'feature_based'
- **selection_type** (*str*) – Type of selection - 'PerClass' | 'Supervised' | 'Full'

lazy_greedy_max(*budget*)

Data selection method using different submodular optimization functions.

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

Returns **total_greedy_list** – List containing indices of the best datapoints

Return type list

1.2.4 Similarity Matrix

class distil.utils.similarity_mat.**SimilarityComputation**(*device*, *x_trn*, *y_trn*, *N_trn*, *batch_size*)

Bases: object

Implementation of Submodular Function. This class allows you to use different submodular functions

Parameters

- **device** (*str*) – Device to be used, cpu|gpu
- **x_trn** (*torch tensor*) – Data on which submodular optimization should be applied
- **y_trn** (*torch tensor*) – Labels of the data
- **model** (*class*) – Model architecture used for training
- **N_trn** (*int*) – Number of samples in dataset
- **batch_size** (*int*) – Batch size to be used for optimization
- **if_convex** (*bool*) – If convex or not
- **submod** (*str*) – Choice of submodular function - 'facility_location' | 'graph_cut' | 'saturated_coverage' | 'sum_redundancy' | 'feature_based'
- **selection_type** (*str*) – Type of selection - 'PerClass' | 'Supervised' | 'Full'

compute_score(*idxs*)
Compute the score of the indices. :param model_params: Python dictionary object containing models parameters :type model_params: OrderedDict :param idxs: The indices :type idxs: list

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_index(*data*, *data_sub*)
Returns indexes of the rows.

Parameters

- **data** (*numpy array*) – Array to find indexes from
- **data_sub** (*numpy array*) – Array of data points to find indexes for

Returns **greedyList** – List of indexes

Return type list

1.3 distil.utils.models package

We have incorporated several neural network architectures in the DISTIL repository. Below given is a list of Neural network ar

- densenet
- dla
- dla_simple
- dpn
- efficientnet
- googlenet
- lenet
- mobilenet
- mobilenetv2
- pnasnet
- preact_resnet
- regnet
- resnet
- resnext
- senet

- shufflenet
- shufflenetv2
- vgg

To use custom model architecture, modify the model architecture in the following way:

The forward method should have two more variables:

1. A boolean variable *last* which -

   *If *true*: returns the model output and the output of the second last layer

   *If *false*: Returns the model output.

2. A boolean variable 'freeze' which -

   *If *true*: disables the tracking of any calculations required to later calculate a gradient i.e skips gradient calculation over the weights

   *If *false*: otherwise

3. get_embedding_dim() method which returns the number of hidden units in the last layer.

CHAPTER

TWO

CONFIGURATION FILES FOR TRAINING

This page gives a tutorial on how to generate your custom training configuration files.

This configuration files can be used to select datasets, training configuration, and active learning settings. These files are in json format.

```
{
        "model": {
                "architecture": "resnet18",
                "target_classes": 10
        },
        "train_parameters": {
                "lr": 0.001,
                "batch_size": 1000,
                "n_epoch": 50,
                "max_accuracy": 0.95,
                "isreset": true,
                "islogs":  true,
                "logs_location": "./logs.txt"
        },

        "active_learning":{
                "strategy": "badge",
                "budget": 1000,
                "rounds": 15,
                "initial_points":1000,

                "strategy_args":{
                        "batch_size" : 1000,
                        "lr":0.001
                }
        },
        "dataset":{
                "name":"cifar10"
        }
}
```

The configuration files consists of following sections:

1. Model
2. Training Parameters
3. Active Learning Configuration
4. Dataset

Symbol (%) represents mandatory arguments

model

1. **architecture %**
    - **Model architecture to be used, Presently it supports the below mentioned architectures.**
        1. resnet18
        2. two_layer_net
2. **target_classes %**
    - Number of output classes for prediction.
3. **input_dim**
    - Input dimension of the dataset. To be mentioned while using two layer net.
4. **hidden_units_1**
    - Number of hidden units to be used in the first layer. To be mentioned while using two layer net.

train_parameters

1. **lr %**
    - Learning rate to be used for training.
2. **batch_size %**
    - Batch size to be used for training.
3. **n_epoch %**
    - Maximum number of epochs for the model to train.
4. **max_accuracy**
    - Maximum training accuracy after which training should be stopped.
5. **isreset**
    - **Reset weight whenever the model training starts.**
        1. True
        2. False
6. **islogs**
    - **Log training output.**
        1. True
        2. False
7. **logs_location %**
    - Location where logs should be saved.

active_learning

1. **strategy %**
    - **Active learning strategy to be used.**
        1. badge
        2. glister
        3. entropy_sampling

4. margin_sampling
5. least_confidence
6. core_set
7. random_sampling
8. fass
9. bald_dropout
10. adversarial_bim
11. kmeans_sampling
12. baseline_sampling
13. adversarial_deepfool

2. **budget %**
    - Number of points to be selected by the active learning strategy.
3. **rounds %**
    - Total number of rounds to run active learning for.
4. **initial_points**
    - Initial number of points to start training with.
5. **strategy_args**
    - Arguments to pass to the strategy. It varies from strategy to strategy. Please refer to the documentation of the strategy that is being used.

dataset

1. **name**
    - **Name of the dataset to be used. It presently supports following datasets.**
        1. cifar10
        2. mnist
        3. fmnist
        4. svhn
        5. cifar100
        6. satimage
        7. ijcnn1

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

PYTHON MODULE INDEX

d

INDEX

E

F

G

K

L

M

R

S