baal Documentation

Release 1.3.0

Parmida Atighehchian, Frédéric Branchaud-Charron, Jan Freyberg

Jun 21, 2021
LEARN MORE ABOUT BAAL

1 Support 3
2 Indices and tables 77
Index 79
Star

BaaL is a Bayesian active learning library. We provide methods to estimate sampling from the posterior distribution in order to maximize the efficiency of labelling during active learning. Our library is suitable for research and industrial applications.

To know more on what is Bayesian active learning, see our User guide.

We are a member of Pytorch’s ecosystem, and we welcome contributions from the community. If you have any question, we are reachable on Gitter.
For support, we have several ways to help you:
- Our FAQ
- Submit an issue on Github here
- Join our Gitter!

1.1 User guide

Welcome to BaaL’s user guide where we will learn on bayesian deep learning and bayesian active learning.

In addition, we propose a cheat sheet that will help users translate an equation to the equivalent BaaL representation.

1.1.1 Notations and glossary

- Training dataset $D_L$
- Pool, the unlabelled portion of the dataset $D_U$
- Heuristic, the function that computes the uncertainty (ex. BALD) $U$
- Active learning step, the sequence of training, selecting and labelling one or many examples.
- BALD, an heuristic that works well with deep learning models that are overconfident.
- Query size, the number of items to label between retraining.
- Iterations, number of Monte Carlo sampling to do.

1.1.2 Active learning

Active learning is a field of machine learning that reduces labelling cost by only labelling the most informative examples. Datasets, especially in industry, contain many similar examples that would bring no information to the model.

To select the next example to label, we first train a machine learning model on the trained dataset. Then we compute the model’s uncertainty on all unlabelled examples. The most uncertain is selected to be labelled.
1.1.3 Bayesian active learning

Bayesian active learning builds upon active learning by framing the problem from a Bayesian point-of-view. In this case, we want to reduce the epistemic uncertainty (i.e. the model’s uncertainty) on a dataset.

In addition, we will do this by sampling from the posterior distribution allowing us to better estimate the uncertainty. As an example, it is common to use MC-Dropout (Gal and Ghahramani, 2016) and BALD (Houlsby et al. 2013) to do this. The former allows us to draw from the posterior distribution and the latter estimates the mutual information. In recent years, new approaches were suggested to improve BALD such as BatchBALD (Kirsch et al, 2019) or ICAL (Jain et al. 2020), but they work on similar principles.

1.1.4 Open challenges

Active learning is a challenging field, many techniques work only on classification or are sensitive to the data distribution. Often, uniform selection sets a strong baseline, especially on academic datasets.

Consequences of using AL

The effect of using active learning is an understudied problem.

While we know that AL creates more balanced datasets, better calibrated models and such. We do not know what is the effect of sampling bias in all settings.

At ICLR 2020, Farquhar et al. showed that sampling bias produces biased estimators, and they propose a new unbiased estimator that gets good results on simple models. We hope that work in this area continues so that we can better understand the impact of active learning.

Resources

- Literature review
- Active learning dataset and training loop classes
- Methods for approximating bayesian posteriors
- Full active learning example

References

1.2 Active learning infrastructure objects

Active learning, or interactively choosing datapoints to request labels for, presents a challenge that requires some data handling infrastructure that’s slightly different to the normal pytorch dataset classes. In particular, a dataset is no longer a static thing, but instead grows as you progress through your experiment or your application.

To handle these needs, baal contains:

- `ActiveLearningDataset`, which is a pytorch dataset that lets you interactively label data.
- `ActiveLearningLoop`, which wraps an `ActiveLearningDataset` and simplifies experiments by allowing you to call `step` whenever you want to label data.

The `ActiveLearningDataset` wraps another pytorch dataset. For an example on how to use it, you can take a look at how we turn the MNIST dataset into an active learning dataset:

```python
[2]: path = "~/Users/jan/datasets/mnist/"

[4]: from torchvision import transforms, datasets
    from baal.active.dataset import ActiveLearningDataset

    transform = transforms.Compose([transforms.Grayscale(3), transforms.ToTensor()])
    test_transform = transform

    active_mnist = ActiveLearningDataset(
        datasets.MNIST(path, train=True, transform=transform),
        pool_specifics={'transform': test_transform},
    )
```

As you can see, this is a fairly thin wrapper around MNIST. But, we can now check several new properties of this dataset:

```python
[5]: active_mnist.n_labelled
[5]: tensor(0)

[6]: active_mnist.n_unlabelled
[6]: tensor(60000)
```

We can also start labelling data. Either randomly, or based on specific indices:

```python
[7]: active_mnist.label_randomly(10)
    active_mnist.label([55, 56, 50100])
```

We’ve just labelled 10 points randomly, and 3 points based on specific indices. Now, if we check how many have been labelled, we see that 13 have been labelled:

```python
[8]: active_mnist.n_labelled
[8]: tensor(13)

[9]: active_mnist.n_unlabelled
[9]: tensor(59987)
```
We will also see that when we check the length of this dataset - something that is done by e.g. pytorch DataLoader classes - it only gives the length of the labelled dataset:

```
[12]: len(active_mnist)
[12]: 13
```

And, if we try to access an item, it will only allow us to index the labelled datapoints:

```
[13]: active_mnist[0]
[13]: (tensor([[0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.],
            ...,  
            [0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.]],
        [[0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.],
            ...,  
            [0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.]], 8))
```

```
[14]: active_mnist[15]
Traceback (most recent call last)
<ipython-input-14-2413857e337e> in <module>
----> 1 active_mnist[15]
~/projects/baal/src/baal/active/dataset.py in __getitem__(self, index)
      30     def __getitem__(self, index: int) -> Tuple[torch.Tensor, ...]:
      31         """Return stuff from the original dataset.""
 ---> 32             return self._dataset[self._labelled_to_oracle_index(index)]
      33     
      34 def __len__(self) -> int:

~/projects/baal/src/baal/active/dataset.py in _labelled_to_oracle_index(self, index)
      54     def _labelled_to_oracle_index(self, index: int) -> int:
      55     """Return labelled to oracle index."
 ---> 56             return self._labelled.nonzero()[index].squeeze().item()
      57     
      58 def _pool_to_oracle_index(self, index: Union[int, List[int]])
```
Instead, if we want to actually use the unlabeled data, we need to use the pool attribute of the active learning dataset, which is itself a dataset:

```python
[15]: len(active_mnist.pool)
[15]: 59987
```

1.3 Methods for approximating bayesian posteriors

When we started developing active learning methods, we realised that what we wanted to achieve required estimating the uncertainty of models. Doing so for neural networks is an ongoing active research area.

For the purposes of baal, we have implemented a few methods that are relatively generic and work with many neural networks.

All the techniques implemented effectively produce approximate samples from the posterior. For classification techniques, this means that you usually end up with a 3D tensor rather than a 2D tensor (n_batch x n_classes x n_samples rather than n_batch x n_classes).

1.3.1 Monte-Carlo Dropout

Monte-Carlo Dropout, or MC Dropout, is a very simple way of accessing uncertainty in a network that include Dropout layers. Essentially, rather than turning off dropout during inference, you keep in on and make multiple predictions on the same data. Due to the stochastic zeroing of weights, you’ll get a different for every iteration, even if the input is the same.

This is valid primarily because you trained the network using dropout: You have already learnt to make predictions without all the weights.

The output is a distribution of predictions.

Usage

In order to use it, you can simply import Dropout layers from baal and use them in your model construction:

```python
[1]: import torch
    import baal.bayesian.dropout

standard_model = torch.nn.Sequential(
    torch.nn.Linear(10, 8),
    torch.nn.ReLU(),
    torch.nn.Dropout(p=0.5),
    torch.nn.Linear(8, 4),
    torch.nn.ReLU(),
    torch.nn.Dropout(p=0.5),
    torch.nn.Linear(4, 2),
)
```

mc_dropout_model = torch.nn.Sequential(
    torch.nn.Linear(10, 8),
    torch.nn.ReLU(),
    baal.bayesian.dropout.Dropout(p=0.5),
    torch.nn.Linear(8, 4),
    torch.nn.ReLU(),
    baal.bayesian.dropout.Dropout(p=0.5),
    torch.nn.Linear(4, 2),
)

The main difference between these is that the standard model will set the dropout probability to zero during eval, while the MC dropout model will not:

[2]:

dummy_input = torch.randn(8, 10)

standard_model.eval()
print(bool((standard_model(dummy_input) == standard_model(dummy_input)).all()))

mc_dropout_model.eval()
print(bool((mc_dropout_model(dummy_input) == mc_dropout_model(dummy_input)).all()))

True
False

In order to get a distribution of model outputs, you simply need to repeatedly run the same data through the MC Dropout model. *baal* makes this easier for you by providing a class called *ModelWrapper*. This class accepts your model and a criterion (loss) function, and provides several utility functions, such as running training steps and more. The one that is important for obtaining a posterior distribution is *ModelWrapper.predict_on_batch*.

This method allows you to specify a number of iterations to run the model for, and produces a distribution accordingly:

[3]:

from baal.modelwrapper import ModelWrapper

wrapped_model = ModelWrapper(
    mc_dropout_model,
    torch.nn.MSELoss()
)

with torch.no_grad():
    predictions = wrapped_model.predict_on_batch(dummy_input, iterations=10000)

The tensor “prediction_distribution” has the shape (batch size) x (output size) x iterations:

[4]:
predictions.shape

[4]:
torch.Size([8, 2, 10000])

We can visualise this posterior distribution, for example for the first data point in our minibatch (although note that because this model is overly simplistic, this is not very useful):

[5]:

import matplotlib.pyplot as plt
%matplotlib inline
1.3.2 Drop Connect

DropConnect is another way of accessing uncertainty in a network. The idea is very similar to MCdropout, however in Dropconnect the connection weights between layers are randomly chosen to be dropped. With multiple passes through the network, similar to MCDropout we will end up with a distribution on the predictions.

Eventually, the benefit of this approach is that you do not need to add a Dropout layer to a network which doesn’t have one. In long term using DropConnect is going to surpass MCDropout results in active learning but the drawbacks is the essential need of almost twice iterations and longer trainings for the model to converge at each active learning step.

**Usage**

As usual we have pre-implemented wrappers to ease your job for this. Example below shows how to use this module:

```python
import torch
class DummyModel(torch.nn.Module):
    def __init__(self):
        super(DummyModel, self).__init__()
        self.conv = torch.nn.Conv2d(3, 8, kernel_size=10)
        self.relu = torch.nn.ReLU()
        self.linear = torch.nn.Linear(8, 1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = self.conv(x)
        x = self.relu(x)
        x = x.view(x.shape[0], -1)
        x = self.linear(x)
        x = self.sigmoid(x)
        return x
```

[6]:
As part of our experiments, we run MCDropout(MCD) and DropConnect(MCDC) against each other. We let the experiments run for 2000 epochs on vgg16 using CIFAR10 and tried different number of iterations and weight drop rate for Dropconnect. As the experiments show, DropConnect could give a better result if it is used with number of iterations more than 80 and drop weight rate of around 50%. The reference paper indicates having a drop rate of 94% should give the best result but our experiments show otherwise. The main factor of change for DropConnect is the number of iterations used to estimate the posterior. However, as we can see for MCDropout, number of iterations 40 and 80 would give almost same results which would overfit by time. In order to prevent overfitting, we could change learning rate and use other techniques and get a lift on the performance, however as could be seen for higher iterations, DropConnect could easily outperform MCDropout at 10K training set size. Finally, the choice of method and training process is always for the user and depending on the problem in hand. Parameters like how low the validation error should be and if the training is allowed to be run for few days or there is a time limit could all effect in which strategy is best and which hyperparameters to choose.
1.4 API Reference

1.4.1 ModelWrapper

class `baal.ModelWrapper(model, criterion, replicate_in_memory=True)`

Wrapper created to ease the training/testing/loading.

Parameters

- **model** (*nn.Module*) – The model to optimize.
- **criterion** (*Callable*) – A loss function.
- **replicate_in_memory** (*bool*) – Replicate in memory optional.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_metric(name, initializer)</code></td>
<td>Add a <code>baal.utils.metric.Metric</code> to the Model.</td>
</tr>
<tr>
<td><code>eval()</code></td>
<td>Set the model in <code>eval mode</code>.</td>
</tr>
<tr>
<td><code>get_params()</code></td>
<td>Return the parameters to optimize.</td>
</tr>
<tr>
<td><code>load_state_dict(state_dict[, strict])</code></td>
<td>Load the model with <code>state_dict</code>.</td>
</tr>
<tr>
<td><code>predict_on_batch(data[, iterations, cuda])</code></td>
<td>Get the model’s prediction on a batch.</td>
</tr>
<tr>
<td><code>predict_on_dataset(dataset, batch_size, ...)</code></td>
<td>Use the model to predict on a dataset <code>iterations</code> time.</td>
</tr>
</tbody>
</table>

continues on next page
### add_metric(name: str, initializer: Callable)
Add a baal.utils.metric.Metric to the Model.

**Parameters**

- **name** (`str`) – name of the metric.
- **initializer** (`Callable`) – lambda to initialize a new instance of a baal.utils.metrics.Metric object.

### eval()
Set the model in eval mode.

### get_params()
Return the parameters to optimize.

**Returns**
Config for parameters.

### load_state_dict(state_dict, strict=True)
Load the model with *state_dict*.

### predict_on_batch(data, iterations=1, cuda=False)
Get the model’s prediction on a batch.

**Parameters**

- **data** (`Tensor`) – The model input.
- **iterations** (`int`) – Number of prediction to perform.
- **cuda** (`bool`) – Use CUDA or not.

**Returns**
Tensor, the loss computed from the criterion. shape = {batch_size, nclass, n_iteration}.

**Raises**
RuntimeError if CUDA runs out of memory during data replication.

### predict_on_dataset(dataset: torch.utils.data.Dataset, batch_size: int, iterations: int, use_cuda: bool, workers: int = 4, collate_fn: Optional[Callable] = None, half=False)
Use the model to predict on a dataset iterations time.

**Parameters**

- **dataset** (`Dataset`) – Dataset to predict on.
- **batch_size** (`int`) – Batch size to use during prediction.
• **iterations** (int) – Number of iterations per sample.
• **use_cuda** (bool) – Use CUDA or not.
• **workers** (int) – Number of workers to use.
• **collate_fn** *(Optional[Callable]*) – The collate function to use.
• **half** (bool) – If True use half precision.

**Notes**

The “batch” is made of \(batch\_size * iterations\) samples.

**Returns** Array \([n\_samples, n\_outputs, ..., n\_iterations]\).

**predict_on_dataset_generator** *(dataset: torch.utils.data.Dataset, batch_size: int, iterations: int, use_cuda: bool, workers: int = 4, collate_fn: Optional[Callable] = None, half=False)*

Use the model to predict on a dataset \(iterations\) time.

**Parameters**

• **dataset** *(Dataset)* – Dataset to predict on.
• **batch_size** (int) – Batch size to use during prediction.
• **iterations** (int) – Number of iterations per sample.
• **use_cuda** (bool) – Use CUDA or not.
• **workers** (int) – Number of workers to use.
• **collate_fn** *(Optional[Callable]*) – The collate function to use.
• **half** (bool) – If True use half precision.

**Notes**

The “batch” is made of \(batch\_size * iterations\) samples.

**Returns** Generators \([batch\_size, n\_classes, ..., n\_iterations]\).

**reset_all()**
Reset all \(resetable\) layers.

**reset_fcs()**
Reset all torch.nn.Linear layers.

**state_dict()**
Get the state dict(s).

**test_on_batch** *(data: torch.Tensor, target: torch.Tensor, cuda: bool = False, average_predictions: int = 1)*

Test the current model on a batch.

**Parameters**

• **data** *(Tensor)* – The model input.
• **target** *(Tensor)* – The ground truth.
• **cuda** (bool) – Use CUDA or not.
• **average_predictions** (int) – The number of predictions to average to compute the test loss.
**Returns**  Tensor, the loss computed from the criterion.

**test_on_dataset**

```python
dataset: torch.utils.data.Dataset, batch_size: int, use_cuda: bool, workers: int = 4, collate_fn: Optional[Callable] = None, average_predictions: int = 1
```

Test the model on a Dataset `dataset`.

**Parameters**
- `dataset` *(Dataset)* – Dataset to evaluate on.
- `batch_size` *(int)* – Batch size used for evaluation.
- `use_cuda` *(bool)* – Use Cuda or not.
- `workers` *(int)* – Number of workers to use.
- `collate_fn` *(Optional[Callable]*) – The collate function to use.
- `average_predictions` *(int)* – The number of predictions to average to compute the test loss.

**Returns**  Average loss value over the dataset.

**train()**

Set the model in *train* mode.

**train_and_test_on_datasets**

```python
```

Train and test the model on both Dataset `train_dataset`, `test_dataset`.

**Parameters**
- `train_dataset` *(Dataset)* – Dataset to train on.
- `test_dataset` *(Dataset)* – Dataset to evaluate on.
- `optimizer` *(Optimizer)* – Optimizer to use during training.
- `batch_size` *(int)* – Batch size used.
- `epoch` *(int)* – Number of epoch to train on.
- `use_cuda` *(bool)* – Use Cuda or not.
- `workers` *(int)* – Number of workers to use.
- `collate_fn` *(Optional[Callable]*) – The collate function to use.
- `regularizer` *(Optional[Callable]*) – The loss regularization for training.
- `return_best_weights` *(bool)* – If True, will keep the best weights and return them.
- `patience` *(Optional[int]*) – If provided, will use early stopping to stop after *patience* epoch without improvement.
- `min_epoch_for_es` *(int)* – Epoch at which the early stopping starts.

**Returns**  History and best weights if required.

**train_on_batch**

```python
data, target, optimizer, cuda=False, regularizer: Optional[Callable] = None
```

Train the current model on a batch using `optimizer`.

**Parameters**
- `data` *(Tensor)* – The model input.
• **target** (Tensor) – The ground truth.
• **optimizer** (optim.Optimizer) – An optimizer.
• **cuda** (bool) – Use CUDA or not.
• **regularizer** (Optional[Callable]) – The loss regularization for training.

Returns Tensor, the loss computed from the criterion.

**train_on_dataset**(*dataset, optimizer, batch_size, epoch, use_cuda, workers=4, collate_fn: Optional[Callable] = None, regularizer: Optional[Callable] = None*)
Train for epoch epochs on a Dataset `dataset`.

Parameters

• **dataset** (Dataset) – Pytorch Dataset to be trained on.
• **optimizer** (optim.Optimizer) – Optimizer to use.
• **batch_size** (int) – The batch size used in the DataLoader.
• **epoch** (int) – Number of epoch to train for.
• **use_cuda** (bool) – Use cuda or not.
• **workers** (int) – Number of workers for the multiprocessing.
• **collate_fn** (Optional[Callable]) – The collate function to use.
• **regularizer** (Optional[Callable]) – The loss regularization for training.

Returns The training history.

### 1.4.2 Active learning functionality

**class** baal.active.**ActiveLearningDataset**(*args: Any, **kwargs: Any*)
A dataset that allows for active learning.

Parameters

• **dataset** (torch.data.Dataset) – The baseline dataset.
• **labelled** (Union[np.ndarray, torch.Tensor]) – An array/tensor that acts as a boolean mask which is True for every data point that is labelled, and False for every data point that is not labelled.
• **make_unlabelled** (Callable) – The function that returns an unlabelled version of a datum so that it can still be used in the DataLoader.
• **random_state** (None, int, RandomState) – Set the random seed for label_randomly().
• **pool_specifics** (Optional[Dict]) – Attributes to set when creating the pool. Useful to remove data augmentation.

Attributes

• **n_labelled** The number of labelled data points.
• **n_unlabelled** The number of unlabelled data points.
• **pool** Returns a new Dataset made from unlabelled samples.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ActiveIter(aldataset)</code></td>
<td>Iterator over an ActiveLearningDataset.</td>
</tr>
<tr>
<td><code>__call__(*args, **kwargs)</code></td>
<td>Call self as a function.</td>
</tr>
<tr>
<td><code>check_dataset_can_label()</code></td>
<td>Check if a dataset can be labelled.</td>
</tr>
<tr>
<td><code>get_raw(idx)</code></td>
<td>Get a datapoint from the underlying dataset.</td>
</tr>
<tr>
<td><code>is_labelled(idx)</code></td>
<td>Check if a datapoint is labelled.</td>
</tr>
<tr>
<td><code>label(index[, value])</code></td>
<td>Label data points.</td>
</tr>
<tr>
<td><code>label_randomly(n)</code></td>
<td>Label n data-points randomly.</td>
</tr>
<tr>
<td><code>load_state_dict(state_dict)</code></td>
<td>Load the labelled map and random_state with give state_dict.</td>
</tr>
<tr>
<td><code>reset_labeled()</code></td>
<td>Reset the label map.</td>
</tr>
<tr>
<td><code>state_dict()</code></td>
<td>Return the state_dict, ie.</td>
</tr>
</tbody>
</table>

```python
class ActiveIter(aldataset):
    # Iterator over an ActiveLearningDataset.

def check_dataset_can_label()
    # Check if a dataset can be labelled.
    # Returns Whether the dataset’s label can be modified or not.
```

Notes

To be labelled, a dataset needs a method `label` with definition: `label(self, idx, value)` where `value` is the label for indice `idx`.

```python
def get_raw(idx: int) → None
    # Get a datapoint from the underlying dataset.

def is_labelled(idx: int) → bool
    # Check if a datapoint is labelled.

def label(index: Union[list, int], value: Optional[Any] = None) → None
    # Label data points. The index should be relative to the pool, not the overall data.

    Parameters
    - `index` (Union[list, int]) – one or many indices to label.
    - `value` (Optional[Any]) – The label value. If not provided, no modification to the underlying dataset is done.

def label_randomly(n: int = 1) → None
    # Label n data-points randomly.

    Parameters
    - `n` (int) – Number of samples to label.
```

```python
def load_state_dict(state_dict)
    # Load the labelled map and random_state with give state_dict.

def n_labelled
    # The number of labelled data points.

def n_unlabelled
    # The number of unlabelled data points.
```

```python
property pool: torch.utils.data.Dataset
    # Returns a new Dataset made from unlabelled samples.
```
 Raises ValueError if a pool specific attribute cannot be set. –

 reset_labeled()
 Reset the label map.

 state_dict()
 Return the state_dict, ie. the labelled map and random_state.

 class baal.active.ActiveLearningLoop(dataset: baal.active.dataset.ActiveLearningDataset,
   get_probabilities: Callable, heuristic:
   baal.active.heuristics.heuristics.AbstractHeuristic =
   <baal.active.heuristics.heuristics.Random object>, ndata_to_label:
   int = 1, max_sample=-1, **kwargs)

 Object that perform the active learning iteration.

 Parameters

 - dataset (ActiveLearningDataset) – Dataset with some sample already labelled.
 - get_probabilities (Function) – Dataset -> **kwargs -> ndarray [n_samples,
   n_outputs, n_iterations].
 - heuristic (Heuristic) – Heuristic from baal.active.heuristics.
 - ndata_to_label (int) – Number of sample to label per step.
 - max_sample (int) – Limit the number of sample used (-1 is no limit).
 - **kwargs – Parameters forwarded to get_probabilities.

 Methods

 step([pool]) Perform an active learning step.

 step(pool=None) → bool
 Perform an active learning step.

 Parameters pool (iterable) – dataset pool indices.

 Returns boolean, Flag indicating if we continue training.

 class baal.active.FileDataset(*args: Any, **kwargs: Any)
 Dataset object that load the files and apply a transformation.

 Parameters

 - files (List[str]) – The files.
 - lbs (List[Any]) – The labels, -1 indicates that the label is unknown.
 - transform (Optional[Callable]) – torchvision.transform pipeline.
 - target_transform (Optional[Callable]) – Function that modifies the target.
 - image_load_fn (Optional[Callable]) – Function that loads the image, by default uses PIL.
 - seed (Optional[int]) – Will set a seed before and between DA.
Methods

__call__(*args, **kwargs) Call self as a function.

label(idx, lbl) Label the sample idx with lbl.

get_kwargs

label(idx: int, lbl: Any)
Label the sample idx with lbl.

Parameters

• idx (int) – The sample index.
• lbl (Any) – The label to assign.

1.4.3 Calibration Wrapper


Adding a linear layer to a classifier model after the model is trained and train this new layer until convergence. Together with the linear layer, the model is now calibrated. Source: https://arxiv.org/abs/1910.12656 Code inspired from: https://github.com/dirichletcal/experiments_neurips

References

@article{kullbeyond,
  title={Beyond temperature scaling: Obtaining well-calibrated multi-class probabilities with Dirichlet calibration Supplementary material},
  author={Kull, Meelis and Perello-Nieto, Miquel and K{"a}ngsepp, Markus and Silva Filho, Telmo and Song, Hao and Flach, Peter}
}

Parameters

• wrapper (ModelWrapper) – Provides training and testing methods.
• num_classes (int) – Number of classes in classification task.
• lr (float) – Learning rate.
• reg_factor (float) – Regularization factor for the linear layer weights.
• mu (float) – Regularization factor for the linear layer biases. If not given, will be initialized by “l”.

Attributes

• calibrated_model
• metrics
### Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>calibrate(train_set, test_set, batch_size,...)</code></td>
<td>Training the linear layer given a training set and a validation set.</td>
</tr>
<tr>
<td><code>l2_reg()</code></td>
<td>Using trainable layer's parameters for l2 regularization.</td>
</tr>
</tbody>
</table>

**calibrate**

```python
```

Training the linear layer given a training set and a validation set. The training set should be different from what model is trained on.

**Parameters**

- **train_set** (`Dataset`) – The training set.
- **test_set** (`Dataset`) – The validation set.
- **batch_size** (`int`) – Batch size used.
- **epoch** (`int`) – Number of epochs to train the linear layer for.
- **use_cuda** (`bool`) – If “True”, will use GPU.
- **double_fit** (`bool`) – If “True” would fit twice on the train set.
- **kwargs** (`dict`) – Rest of parameters for baal.ModelWrapper.train_and_test_on_dataset().

**Returns** List of loss values for each epoch. model.state_dict (dict): Model weights.

**Return type** loss_history (list[float])

**l2_reg()**

Using trainable layer’s parameters for l2 regularization.

**Returns** The regularization term for the linear layer.

### 1.4.4 Heuristics

**class** baal.active.heuristics.AbstractHeuristic(shuffle_prop=0.0, reverse=False, reduction='none')

Abstract class that defines a Heuristic.

**Parameters**

- **shuffle_prop** (`float`) – shuffle proportion.
- **reverse** (`bool`) – True if the most uncertain sample has the highest value.
- **reduction** (`Union[str, Callable]`) – Reduction used after computing the score.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(predictions)</td>
<td>Rank the predictions according to their uncertainties.</td>
</tr>
<tr>
<td>compute_score(predictions)</td>
<td>Compute the score according to the heuristic.</td>
</tr>
<tr>
<td>get_ranks(predictions)</td>
<td>Rank the predictions according to their uncertainties.</td>
</tr>
<tr>
<td>get_uncertainties(predictions)</td>
<td>Get the uncertainties.</td>
</tr>
<tr>
<td>get_uncertainties_generator(predictions)</td>
<td>Compute the score according to the heuristic.</td>
</tr>
<tr>
<td>reorder_indices(scores)</td>
<td>Order indices given their uncertainty score.</td>
</tr>
</tbody>
</table>

**compute_score(predictions)**

Compute the score according to the heuristic.

- **Parameters** predictions (ndarray) – Array of predictions
- **Returns** Array of scores.

**get_ranks(predictions)**

Rank the predictions according to their uncertainties.

- **Parameters** predictions (ndarray) – [batch_size, C, ..., Iterations]
- **Returns** Ranked index according to the uncertainty (highest to lowest).

**get_uncertainties(predictions)**

Get the uncertainties.

- **Parameters** predictions (ndarray) – Array of predictions
- **Returns** Array of uncertainties

**get_uncertainties_generator(predictions)**

Compute the score according to the heuristic.

- **Parameters** predictions (Iterable) – Generator of predictions
- **Raises** ValueError if the generator is empty.
- **Returns** Array of scores.

**reorder_indices(scores)**

Order indices given their uncertainty score.

- **Parameters** scores (ndarray/ List[ndarray]) – Array of uncertainties or list of arrays.
- **Returns** ordered index according to the uncertainty (highest to lowest).
- **Raises** ValueError if scores is not uni-dimensional.

**class baal.active.heuristics.BALD(shuffle_prop=0.0, reduction='none')**

Sort by the highest acquisition function value.

- **Parameters**
  - **shuffle_prop** (float) – Amount of noise to put in the ranking. Helps with selection bias (default: 0.0).
  - **reduction** (Union[str, callable]) – function that aggregates the results (default: 'none').
References

https://arxiv.org/abs/1703.02910

Methods

```python
__call__(predictions) Rank the predictions according to their uncertainties.
compute_score(predictions) Compute the score according to the heuristic.
get_ranks(predictions) Rank the predictions according to their uncertainties.
get_uncertainties(predictions) Get the uncertainties.
get_uncertainties_generator(predictions) Compute the score according to the heuristic.
reorder_indices(scores) Order indices given their uncertainty score.
```

class baal.active.heuristics.Random

Random heuristic.

Parameters

- `shuffle_prop` (float) – UNUSED
- `reduction` (Union[str, callable]) – UNUSED.

Methods

```python
__call__(predictions) Rank the predictions according to their uncertainties.
compute_score(predictions) Compute the score according to the heuristic.
get_ranks(predictions) Rank the predictions according to their uncertainties.
get_uncertainties(predictions) Get the uncertainties.
get_uncertainties_generator(predictions) Compute the score according to the heuristic.
reorder_indices(scores) Order indices given their uncertainty score.
```

class baal.active.heuristics.Entropy

Sort by the highest entropy.

Parameters

- `shuffle_prop` (float) – Amount of noise to put in the ranking. Helps with selection bias (default: 0.0).
- `reduction` (Union[str, callable]) – function that aggregates the results (default: none).

Methods

```python
__call__(predictions) Rank the predictions according to their uncertainties.
compute_score(predictions) Compute the score according to the heuristic.
get_ranks(predictions) Rank the predictions according to their uncertainties.
get_uncertainties(predictions) Get the uncertainties.
get_uncertainties_generator(predictions) Compute the score according to the heuristic.
reorder_indices(scores) Order indices given their uncertainty score.
```
1.4.5 Pytorch Lightning Compatibility

class baal.utils.pytorch_lightning.ResetCallback(*args: Any, **kwargs: Any)
Callback to reset the weights between active learning steps.

Parameters weights (dict) – State dict of the model.

Notes
The weight should be deep copied beforehand.

Methods

on_train_start(trainer, module) Will reset the module to its initial weights.

on_train_start(trainer, module)
Will reset the module to its initial weights.

class baal.utils.pytorch_lightning.BaalTrainer(*args: Any, **kwargs: Any)
Object that perform the training and active learning iteration.

Parameters

• dataset (ActiveLearningDataset) – Dataset with some sample already labelled.
• heuristic (Heuristic) – Heuristic from baal.active.heuristics.
• ndata_to_label (int) – Number of sample to label per step.
• max_sample (int) – Limit the number of sample used (-1 is no limit).
• **kwargs – Parameters forwarded to get_probabilities and to pytorch_lightning
  Trainer.__init__

Methods

predict_on_dataset([model, dataloader]) For documentation, see predict_on_dataset_generator

predict_on_dataset_generator([model, dataloader]) Predict on the pool loader.

predict_on_dataset(model=None, dataloader=None, *args, **kwargs)
For documentation, see predict_on_dataset_generator

predict_on_dataset_generator(model=None, dataloader=None, *args, **kwargs)
Predict on the pool loader.

Parameters

• model – Model to be used in prediction. If None, will get the Trainer’s model.
• dataloader (Optional[DataLoader]) – If provided, will predict on this dataloader.
  Otherwise, uses model.pool_dataloader().

Returns Numpy arrays with all the predictions.
```python
class baal.utils.pytorch_lightning.BaalDataModule(*args: Any, **kwargs: Any)

Methods

pool_dataloader() Create Dataloader for the pool of unlabelled examples.

pool_dataloader() Create Dataloader for the pool of unlabelled examples.

1.5 Baal FAQ

If you have more questions, please submit an issue, and we will include it here!

1.5.1 How to predict uncertainty per sample in a dataset

```python
model = YourModel()
# If not done already, you can wrap your model with our MCDropoutModule
model = MCDropoutModule(model)
dataset = YourDataset()
wrapper = ModelWrapper(model, criterion=None)

heuristic = BALD()

# This has a shape [iterations, len(dataset), num_classes, ...]
predictions = wrapper.predict_on_dataset(dataset, batch_size=32, iterations=20, use_cuda=True)
uncertainty = heuristic.get_uncertainties(predictions)
```

If your model or dataset is too large:

```python
pred_generator = wrapper.predict_on_datasetGenerator(dataset, batch_size=32, iterations=20, use_cuda=True)
uncertainty = heuristic.get_uncertainties_generator(pred_generator)
```

1.5.2 Does BaaL work on semantic segmentation?

Yes! See the example in experiments/segmentation/unet_mcdropout_pascal.py.

The key idea is to provide the Heuristic with a way to aggregate the uncertainties. In the case of semantic segmentation, MC-Dropout will provide a distribution per pixel. To reduce this to a single uncertainty value, you can provide reduction to the Heuristic with one of the following arguments:

- String (one of 'max', 'mean', 'sum')
- Callable, a function that will receive the uncertainty per pixel.
1.5.3 Does BaaL work on NLP/TS/Tabular data?

BaaL is not task-specific, it can be used on a variety of domains and tasks. We are working toward more examples. Bayesian active learning has been used for Text Classification and NER in (Siddhant and Lipton, 2018).

1.5.4 How to know if my model is calibrated

Baal uses the ECE to compute the calibration of a model. It is available through: `baal.utils.metrics.ECE` and `baal.utils.metrics.ECE_PerCLs`, the latter providing the metrics per class.

You can add this metric to your model wrapper doing `ModelWrapper.add_metric('ece', lambda: ECE(n_bins=20))`

After training and testing, you can get your score with:

```python
metrics = your_model.metrics
# Test ECE
metrics['test_ece'].value
# Train ECE
metrics['train_ece'].value
```

1.5.5 What to do if my models/datasets don’t fit in memory?

There is several ways to use Baal on large tasks.

- If MC sampling does not fit, you can use a for-loop instead.
  - Set `ModelWrapper.replicate_in_memory=False`.
- If the size of the prediction does not fit.
  - Heuristics support generators
  - Use `ModelWrapper.predict_on_dataset_generator`

1.5.6 How can I specify that a label is missing and how to label it.

The source of truth for what is labelled is the `ActiveLearningDataset.labelled` array. This means that we will never train on a sample if it is not labelled according to this array. This array determines the split between the labelled and unlabelled datasets.

```python
# Let ds = D, the entire dataset with labelled/unlabelled data.
ds = YourDataset()
al_dataset = ActiveLearningDataset(ds, ...)
# For convenience, let's label 10 samples at random.
# But you can provide the 'labelled' array to ActiveLearningDataset
# if you already have labels.
al_dataset.label_randomly(10)
pool = al_dataset.pool
```

From a rigorous point of view: $D = ds$, $D_L = al\_dataset$ and $D_U = D \setminus D_L = pool$. Then, we train our model on $D_L$ and compute the uncertainty on $D_U$. The most uncertain samples are labelled and added to $D_L$, removed from $D_U$. 

24 Chapter 1. Support
Let a method `query_human` performs the annotations, we can label our dataset using indices relative to $D_U$. This assumes that your dataset class `YourDataset` has a method named `label` which has the following definition: 

```python
def label(self, idx, value)
```

where we give the label for index `idx`. There the index is not relative to the pool, so you don’t have to worry about it.

### Full example.

```python
# Some definitions
your_heuristic = BALD()
pool = active_dataset.pool
your_predictions = ModelWrapper.predict_on_dataset(pool, ...)
# The shape of `your_predictions` is [len(pool), n_classes, ..., iterations]
# Get the next batch of samples to label. Note: These indices are according to the pool.
ranks = your_heuristic(your_predictions)

# Now let's ask a human to label those samples.
labels = query_human(ranks, pool)

# To edit the dataset labels, you can now add those labels to your dataset. Still, the...-indices are according to the pool.
active_dataset.label(ranks, labels)
```

### 1.5.7 Tips & Trick for a successful active learning experiment

Many of these tips can be found in our paper *Bayesian active learning for production*.

**Remove data augmentation when computing uncertainty**

You can specify which variables to override when creating the unlabelled pool using the `pool_specifics` argument.

```python
from torchvision import transforms
transform = transforms.Compose([transform.Resize(32),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor()])
test_transform = transforms.Compose([transform.Resize(32),
    transforms.ToTensor()])

your_dataset = ADataset(transform=transform)
active_dataset = ActiveLearningDataset(your_dataset, pool_specifics={'transform':test_transform})

# active_dataset will use data augmentation
# the pool will use the `test_transform`
pool = active_dataset.pool
```
Reset the model to its original weights (Gal et al. 2017)

```python
# Make a deep copy of the initial weights
initial_weights = copy.deepcopy(model.state_dict())
loop = ActiveLearningLoop(...)

for al_step in range(NUM_AL_STEP):
    # Reset the weights to its initial value
    model.load_state_dict(initial_weights)
    # Train to convergence
    model.train_on_dataset(...)
    # Test on the validation set.
    model.test_on_dataset(...)
    # Label the next set of labels.
    loop.step()
```

Use Bayesian model average when testing.

When using MC-Dropout, or any other Bayesian methods, you will want to compute the Bayesian model average (BMA) at test time too.

To do so, you can specify the `average_predictions` parameters in `ModelWrapper.test_on_dataset`. The prediction will be averaged over iterations stochastic predictions.

This will slightly increase the ECE of your model and will improve the predictive performance as well.

Compute uncertainty on a subset of the unlabelled pool

Predicting on the unlabelled pool is the most time consuming part of active learning, especially in expensive tasks such as segmentation.

Our work shows that predicting on a random subset of the pool is as effective as the full prediction. BaaL supports this features throught the `max_samples` argument in `ActiveLearningPool`.

1.6 How to do research and visualize progress

In this tutorial, we will show how to use BaaL for research ie. when we know the labels. We will introduce notions such as dataset management, MC-Dropout, BALD. If you need more documentation, be sure to check our Additional resources section below!

BaaL can be used on a variety of research domains:

- Active Learning
- Uncertainty estimation
- Fairness, Accountability, Transparency and Ethics (FATE)
- And more!

Today we will focus on a simple example with CIFAR10 and we will animate the progress of active learning!
1.6.1 Requirements

In addition to BaaL standard requirements, you will need to install:

- MulticoreTSNE
- Matplotlib

1.6.2 Additional resources

- More info on the inner working of Active Learning Dataset [here](#).
- To know more about Bayesian deep learning please see our Literature review.

Let's do this!

```python
# Let's start with a bunch of imports.
import random
from copy import deepcopy
from dataclasses import dataclass
import numpy as np
import torch
import torch.backends
from torch import optim
from torch.hub import load_state_dict_from_url
from torch.nn import CrossEntropyLoss
from torchvision import datasets
from torchvision import models
from torchvision.transforms import transforms
from tqdm.autonotebook import tqdm

from baal.active import get_heuristic, ActiveLearningDataset
from baal.active.active_loop import ActiveLearningLoop
from baal.bayesian.dropout import patch_module
from baal.modelwrapper import ModelWrapper

def vgg16(num_classes):
    model = models.vgg16(pretrained=False, num_classes=num_classes)
    weights = load_state_dict_from_url('https://download.pytorch.org/models/vgg16-397923af.pth')
    weights = {k: v for k, v in weights.items() if 'classifier.6' not in k}
    model.load_state_dict(weights, strict=False)
    return model
```

1.6. How to do research and visualize progress
Dataset management and the pool

At many places in our library, we mention the pool, but what is it? The pool is simply the set of unlabelled examples, the example that the model has not deemed important enough to be labelled.

In BaaL, the object that manages this is called the `baal.active.ActiveLearningDataset`. You supply it with your dataset and by default, everything is unlabelled.

```python
al_dataset = ActiveLearningDataset(your_dataset)
```

You can then start with the initial number of examples that are labelled randomly. Let’s label 100 examples.

```python
al_dataset.label_randomly(100)
```

When iterating over `al_dataset`, you will only get the labelled examples. If you need to work on the pool, then you can call `al_dataset.pool` which would return it.

**How can I disable data augmentation when iterating on the pool?**

Disabling data augmentation when computing uncertainty is preferable, we wouldn’t want to disrupt the uncertainty estimation. Fortunately, BaaL can help you with that.

In `ActiveLearningDataset`, we can supply the attribute `pool_specifics` with a dictionary of what to modify in the pool. For example, my `Dataset` has an attribute `transform` which applies the data augmentation. I can modify with:

```python
ActiveLearningDataset(your_dataset, pool_specifics={'transform': test_transform})
```

where `test_transform` is the test version of `transform` without data augmentation.

[2]:

```python
import torch.utils.data as torchdata
class TransformAdapter(torchdata.Subset):
    @property
    def transform(self):
        if hasattr(self.dataset, 'transform'):
            return self.dataset.transform
        else:
            raise AttributeError()
    @transform.setter
    def transform(self, transform):
        if hasattr(self.dataset, 'transform'):
            self.dataset.transform = transform
```

Here we define our Experiment configuration, this can come from your favorite experiment manager like MLFlow. BaaL does not expect a particular format as all arguments are supplied.

[3]:

(continues on next page)
@dataclass
class ExperimentConfig:
    epoch: int = 20000 // 100
    batch_size: int = 32
    initial_pool: int = 512
    query_size: int = 100
    lr: float = 0.001
    heuristic: str = 'bald'
    iterations: int = 40
    training_duration: int = 10

Problem definition

We will perform active learning on a toy dataset, CIFAR-3 where we only keep dogs, cats and airplanes. This will make visualization easier.

```python
[ ]: def get_datasets(initial_pool):
    
    Let's create a subset of CIFAR10 named CIFAR3, so that we can visualize things better.

    We will only select the classes airplane, cat and dog.

    Args:
        initial_pool: Amount of labels to start with.

    Returns:
        ActiveLearningDataset, Dataset, the training and test set.

    # airplane, cat, dog
    classes_to_keep = [0, 3, 5]
    transform = transforms.Compose(
        
        transforms.Resize((32, 32)),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(30),
        transforms.ToTensor(),
        transforms.Normalize(3 * [0.5], 3 * [0.5]),
    )
    test_transform = transforms.Compose(
        
        transforms.Resize((32, 32)),
        transforms.ToTensor(),
        transforms.Normalize(3 * [0.5], 3 * [0.5]),
    )
    train_ds = datasets.CIFAR10('.', train=True,
        transform=transform, target_transform=None,
        #download=True)

    train_mask = np.where([y in classes_to_keep for y in train_ds.targets])[0]
    train_ds = TransformAdapter(train_ds, train_mask)
```

(continues on next page)
# In a real application, you will want a validation set here.

test_set = datasets.CIFAR10(
    '.', train=False,
    transform=test_transform, target_transform=None,
    download=True)

test_mask = np.where([y in classes_to_keep for y in test_set.targets])[0]
test_set = TransformAdapter(test_set, test_mask)

# Here we set 'pool_specifics', where we set the transform attribute for the pool.
active_set = ActiveLearningDataset(train_ds, pool_specifics={'transform': test_transform})

# We start labeling randomly.
active_set.label_randomly(initial_pool)
return active_set, test_set

Creating our experiment

We are now ready to instantiate all of our components:

1. Our `ActiveLearningDataset` and a test dataset.
2. Our heuristic (BALD)

BaaL simplifies your experiments by providing ModelWrapper and ActiveLearningLoop.

- ModelWrapper
- Performs MC sampling efficiently
- Training/testing loops
- ActiveLearningLoop
- Will make prediction on the pool and label the most uncertain examples.

```python
hyperparams = ExperimentConfig()
use_cuda = torch.cuda.is_available()
torch.backends.cudnn.benchmark = True
random.seed(1337)
torch.manual_seed(1337)
if not use_cuda:
    print("warning, the experiments would take ages to run on cpu")

# Get datasets
active_set, test_set = get_datasets(hyperparams.initial_pool)

# Get our model.
heuristic = get_heuristic(hyperparams.heuristic)
criterion = CrossEntropyLoss()
model = vgg16(num_classes=10)

# change dropout layer to MCDropout
model = patch_module(model)
```
if use_cuda:
    model.cuda()
optimizer = optim.SGD(model.parameters(), lr=hyperparams.lr, momentum=0.9)

# Wraps the model into a usable API.
model = ModelWrapper(model, criterion)

# for ActiveLearningLoop we use a smaller batchsize
# since we will stack predictions to perform MCDropout.
active_loop = ActiveLearningLoop(active_set,
                                 model.predict_on_dataset,
                                 heuristic,
                                 hyperparams.query_size,
                                 batch_size=1,
                                 iterations=hyperparams.iterations,
                                 use_cuda=use_cuda)

# We will reset the weights at each active learning step so we make a copy.
init_weights = deepcopy(model.state_dict())

What is an active learning loop

An active learning loop is the process of:

1. Training
2. Estimate uncertainty on the pool
3. Label the most uncertain examples.

```python
[ ]: labelling_progress = active_set._labelled.copy().astype(np.uint16)
for epoch in tqdm(range(hyperparams.epoch)):
    # Load the initial weights.
    model.load_state_dict(init_weights)

    # Train the model on the currently labelled dataset.
    _ = model.train_on_dataset(active_set, optimizer=optimizer, batch_size=hyperparams.
                               batch_size,
                               use_cuda=use_cuda, epoch=hyperparams.training_duration)

    # Get test NLL!
    model.test_on_dataset(test_set, hyperparams.batch_size, use_cuda,
                           average_predictions=hyperparams.iterations)
    metrics = model.metrics

    # We can now label the most uncertain samples according to our heuristic.
    should_continue = active_loop.step()
    # Keep track of progress
    labelling_progress += active_set._labelled.copy().astype(np.uint16)
```

(continues on next page)
if not should_continue:
    break

test_loss = metrics['test_loss'].value
logs = {
    "test_nll": test_loss,
    "epoch": epoch,
    "Next Training set size": len(active_set)
}

100%| 9988/9988 [02:09<00:00, 76.99it/s]
100%| 10888/10888 [02:21<00:00, 77.00it/s]
100%| 11088/11088 [02:23<00:00, 77.10it/s]
100%| 12188/12188 [02:38<00:00, 77.12it/s]
100%| 12988/12988 [02:48<00:00, 76.94it/s]
100%| 13288/13288 [02:52<00:00, 76.96it/s]
100%| 14088/14088 [03:02<00:00, 77.17it/s]
100%| 14488/14488 [03:08<00:00, 76.95it/s]
100%| 14388/14388 [03:06<00:00, 77.12it/s]
100%| 14388/14388 [03:06<00:00, 77.12it/s]
100%| 14288/14288 [03:05<00:00, 76.83it/s]
100%| 14288/14288 [03:05<00:00, 76.83it/s]
100%| 14188/14188 [03:04<00:00, 76.99it/s]
100%| 14188/14188 [03:04<00:00, 76.99it/s]
100%| 14088/14088 [03:03<00:00, 77.00it/s]
100%| 14088/14088 [03:03<00:00, 77.00it/s]
100%| 13988/13988 [03:02<00:00, 77.01it/s]
100%| 13988/13988 [03:02<00:00, 77.01it/s]
100%| 13888/13888 [03:01<00:00, 77.02it/s]
100%| 13888/13888 [03:01<00:00, 77.02it/s]
100%| 13788/13788 [02:59<00:00, 77.03it/s]
100%| 13788/13788 [02:59<00:00, 77.03it/s]
100%| 13688/13688 [02:58<00:00, 77.04it/s]
100%| 13688/13688 [02:58<00:00, 77.04it/s]
100%| 13588/13588 [02:57<00:00, 77.04it/s]
100%| 13588/13588 [02:57<00:00, 77.04it/s]
100%| 13488/13488 [02:56<00:00, 77.04it/s]
100%| 13488/13488 [02:56<00:00, 77.04it/s]
100%| 13388/13388 [02:55<00:00, 77.04it/s]
100%| 13388/13388 [02:55<00:00, 77.04it/s]
100%| 13288/13288 [02:54<00:00, 77.04it/s]
100%| 13288/13288 [02:54<00:00, 77.04it/s]
100%| 13188/13188 [02:53<00:00, 77.04it/s]
100%| 13188/13188 [02:53<00:00, 77.04it/s]
100%| 13088/13088 [02:52<00:00, 77.04it/s]
100%| 13088/13088 [02:52<00:00, 77.04it/s]
We will now save our progress on disk.

```python
model_weight = model.state_dict()
dataset = active_set.state_dict()
torch.save({'model':model_weight, 'dataset':dataset, 'labelling_progress':labelling_progress}, 'checkpoint.pth')
print(model.state_dict().keys(), dataset.keys(), labelling_progress)
```

We will now save our progress on disk.

Visualization

Now that our active learning experiment is completed, we can visualize it!

Get t-SNE features.

We will use MultiCoreTSNE to get a t-SNE representation of our dataset. This will allow us to visualize the progress.

```python
# modify our model to get features
from torch import nn
from torch.utils.data import DataLoader

class FeatureExtractor(nn.Module):
    def __init__(self, model):
        super().__init__()
        self.model = model
    def forward(self, x):
        return torch.flatten(self.model.features(x),1)

features = FeatureExtractor(model.model)
acc = []
for x,y in DataLoader(active_set._dataset, batch_size=10):
    acc.append((features(x.cuda()).detach().cpu().numpy(), y.detach().cpu().numpy()))
xs, ys = zip(*acc)
```

1.6. How to do research and visualize progress
from MulticoreTSNE import MulticoreTSNE as TSNE

# Compute t-SNE on the extracted features.
# Compute t-SNE on the extracted features.
# Compute t-SNE on the extracted features.
tsne = TSNE(n_jobs=4)
transformed = tsne.fit_transform(np.vstack(xs))

labels = np.concatenate(ys)
labels.shape

To make the animation, BaaL has `baal.utils.plot_utils.make_animation_from_data` which takes a set of features, their labels and the array containing the progress we created earlier.

[27]:

```python
from baal.utils.plot_utils import make_animation_from_data

# Create frames to animate the process.
frames = make_animation_from_data(transformed, labels, labelling_progress, ["airplane", "cat", "dog"])
```

[28]:

```python
from IPython.display import HTML
import matplotlib.pyplot as plt
from matplotlib import animation

def plot_images(img_list):
    def init():
        img.set_data(img_list[0])
        return (img,)

    def animate(i):
        img.set_data(img_list[i])
        return (img,)

    fig = plt.Figure(figsize=(10,10))
    ax = fig.gca()
    img = ax.imshow(img_list[0])
    anim = animation.FuncAnimation(fig, animate, init_func=init, frames=len(img_list), interval=60, blit=True)
    return anim

HTML(plot_images(frames).to_jshtml())
```
Conclusion

And that’s it! Using a couple lines of code, we were able to run our active learning experiment and plot the progress on a t-SNE representation.

1.7 Use BaaL in production (Classification)

In this tutorial, we will show you how to use BaaL during your labeling task.

**NOTE** In this tutorial, we assume that we do not know the labels!

1.7.1 Install baal

```
pip install baal
```

We will first need a dataset! For the purpose of this demo, we will use a classification dataset, but BaaL works on more than computer vision! As long as we can estimate the uncertainty of a prediction, BaaL can be used.

We will use the Natural Images Dataset. Please extract the data in /tmp/natural_images.

```
from glob import glob
import os
from sklearn.model_selection import train_test_split
files = glob('/tmp/natural_images/*/*.jpg')
classes = os.listdir('/tmp/natural_images')
train, test = train_test_split(files, random_state=1337)  # Split 75% train, 25%
print(f'Train: {len(train)}, Valid: {len(test)}, Num. classes : {len(classes)}')
```

Train: 5174, Valid: 1725, Num. classes : 8

Introducing baal.active.FileDataset and baal.active.ActiveLearningDataset

FileDataset is simply an object that loads data and implements def label(self, idx: int, lbl: Any). This method is necessary to label items in the dataset. You can set any value you want for unlabelled items, in our example we use -1.

ActiveLearningDataset is a wrapper around a Dataset that performs data management. When you iterate over it, it will return labelled items only.

To learn more on dataset management, visit [this notebook](#).

```
from baal.active import FileDataset, ActiveLearningDataset
from torchvision import transforms
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                      transforms.Resize(224),
                                      transforms.RandomCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])])

# We use -1 to specify that the data is unlabeled.
```

(continues on next page)
train_dataset = FileDataset(train, [-1] * len(train), train_transform)

test_transform = transforms.Compose([transforms.Resize(224),
                                      transforms.RandomCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])])

# We use -1 to specify that the data is unlabeled.
test_dataset = FileDataset(test, [-1] * len(test), test_transform)
active_learning_ds = ActiveLearningDataset(train_dataset, pool_specifics={'transform': test_transform})

We now have two unlabeled datasets: train and validation. We encapsulate the training dataset in an
ActiveLearningDataset object which will take care of the split between labeled and unlabeled samples. We are
now ready to use Active Learning. We will use a technique called MC-Dropout, BaaL supports other techniques (see
README) and proposes a similar API for each of them. When using MC-Dropout with BaaL, you can use any model
as long as there are some Dropout Layers. These layers are essential to compute the uncertainty of the model.
BaaL propose several models, but it also supports custom models using baal.bayesian.dropout.MCDropoutModule.

In this example, we will use VGG-16, a popular model from torchvision.

[3]:
import torch
from torch import nn, optim
from baal.modelwrapper import ModelWrapper
from torchvision.models import vgg16
from baal.bayesian.dropout import MCDropoutModule
USE_CUDA = torch.cuda.is_available()
model = vgg16(pretrained=False, num_classes=len(classes))
# This will modify all Dropout layers to be usable at test time which is
# required to perform Active Learning.
model = MCDropoutModule(model)
if USE_CUDA:
    model.cuda()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9, weight_decay=5e-4)

# ModelWrapper is an object similar to keras.Model.
baal_model = ModelWrapper(model, criterion)
1.7.2 Heuristics

To rank uncertainty, we will use a heuristic. For classification and segmentation, BALD is the recommended heuristic. We will also add noise to the heuristic to lower the selection bias added by the AL process. This is done by specifying shuffle_prop in the heuristic constructor.

```python
from baal.active.heuristics import BALD
heuristic = BALD(shuffle_prop=0.1)
```

1.7.3 Oracle

When the AL process requires a new item to labeled, we need to provide an Oracle. In your case, the Oracle will be a human labeler most likely. For this example, we’re lucky the class label is in the image path!

```python
# This function would do the work that a human would do.
def get_label(img_path):
    return classes.index(img_path.split('/')[2])
```

1.7.4 Labeling process

The labeling will go like this: 1. Label all the test set and some samples from the training set. 2. Train the model for a few epoch on the training set. 3. Select the K-top uncertain samples according to the heuristic. 4. Label those samples. 5. If not done, go back to 2.

```python
import numpy as np
# 1. Label all the test set and some samples from the training set.
for idx in range(len(test_dataset)):
    img_path = test_dataset.files[idx]
    test_dataset.label(idx, get_label(img_path))

# Let's label 100 training examples randomly first.
# Note: the indices here are relative to the pool of unlabelled items!
train_idxs = np.random.permutation(np.arange(len(train_dataset)))[100].tolist()
labels = [get_label(train_dataset.files[idx]) for idx in train_idxs]
active_learning_ds.label(train_idxs, labels)
print(f"Num. labeled: {len(active_learning_ds)}/{len(train_dataset)}")

Num. labeled: 100/5174
```

```python
# 2. Train the model for a few epoch on the training set.
baal_model.train_on_dataset(active_learning_ds, optimizer, batch_size=16, epoch=5, use_cuda=USE_CUDA)
baal_model.test_on_dataset(test_dataset, batch_size=16, use_cuda=USE_CUDA)
print("Metrics:", {k:v.avg for k,v in baal_model.metrics.items()})
```
# 3. Select the K-top uncertain samples according to the heuristic.

```python
pool = active_learning_ds.pool
if len(pool) == 0:
    raise ValueError("We're done!")
```

# We make 15 MCDropout iterations to approximate the uncertainty.
```python
predictions = baal_model.predict_on_dataset(pool, batch_size=16, iterations=15, use_cuda=USE_CUDA)
```

# We will label the 10 most uncertain samples.
```python
top_uncertainty = heuristic(predictions)[:10]
```

# 4. Label those samples.
```python
labels = [get_label(train_dataset.files[idx]) for idx in top_uncertainty]
print(list(zip(labels, top_uncertainty)))
active_learning_ds.label(top_uncertainty, labels)
```

# 5. If not done, go back to 2.
```python
for step in range(5):  # 5 Active Learning step!
    # 2. Train the model for a few epoch on the training set.
    print(f"Training on {len(active_learning_ds)} items!")
    baal_model.train_on_dataset(active_learning_ds, optimizer, batch_size=16, epoch=5, use_cuda=USE_CUDA)
    baal_model.test_on_dataset(test_dataset, batch_size=16, use_cuda=USE_CUDA)

    print("Metrics:", {k:v.avg for k,v in baal_model.metrics.items()})

    # 3. Select the K-top uncertain samples according to the heuristic.
    pool = active_learning_ds.pool
    if len(pool) == 0:
        print("We're done!")
        break
    predictions = baal_model.predict_on_dataset(pool, batch_size=16, iterations=15, use_cuda=USE_CUDA)
    top_uncertainty = heuristic(predictions)[:10]

    # 4. Label those samples.
    labels = [get_label(train_dataset.files[idx]) for idx in top_uncertainty]
    active_learning_ds.label(top_uncertainty, labels)
```
1.7. Use BaaL in production (Classification)
And we're done! Be sure to save the dataset and the model.

```python
[11]: torch.save(
    {'active_dataset': active_learning_ds.state_dict(),
     'model': baal_model.state_dict(),
     'metrics': {k:v.avg for k,v in baal_model.metrics.items()}
    }, '/tmp/baal_output.pth')
```

Support

Submit an issue or reach us to our Gitter!

## 1.8 How to use Deep ensembles in BaaL

Ensemble are one of the easiest form of Bayesian deep learning. The main drawback from this approach is the important amount of computational resources needed to perform it. In this notebook, we will present BaaL's Ensemble API namely `EnsembleModelWrapper`.

This notebook is similar to our notebook on how to do research, we suggest you look at it first if you haven't.

### 1.8.1 Additional resources

- More info on the inner working of Active Learning Dataset [here](#).
- To know more about Bayesian deep learning please see our Literature review.

```python
[8]: import random
    from copy import deepcopy
    from dataclasses import dataclass

    import numpy as np
    import torch.backends
    import optim, nn
    from torch.hub import load_state_dict_from_url
    from torch.nn import CrossEntropyLoss
    from torchvision import datasets
    from torchvision import models
    from torchvision.transforms import transforms
    from tqdm.autonotebook import tqdm

    from baal.active import get_heuristic, ActiveLearningDataset
    from baal.active.active_loop import ActiveLearningLoop
    from baal.ensemble import EnsembleModelWrapper

    def vgg16(num_classes):
        model = models.vgg16(pretrained=False, num_classes=num_classes)
        weights = load_state_dict_from_url('https://download.pytorch.org/models/vgg16-
    ...397923af.pth')
```

(continues on next page)
weights = {k: v for k, v in weights.items() if 'classifier.6' not in k}
model.load_state_dict(weights, strict=False)
return model

def weights_reset(m):
    if isinstance(m, nn.Linear):
        nn.init.normal_(m.weight, 0, 0.01)
        nn.init.constant_(m.bias, 0)

[10]: @dataclass
class ExperimentConfig:
    epoch: int = 15000//256
    batch_size: int = 32
    initial_pool: int = 512
    query_size: int = 100
    lr: float = 0.001
    heuristic: str = 'bald'
    iterations: int = 5 # Set a low number here since each iteration will train a new model.
    training_duration: int = 10

def get_datasets(initial_pool):
    transform = transforms.Compose(
        [transforms.Resize((32, 32)),
         transforms.RandomHorizontalFlip(),
         transforms.RandomRotation(30),
         transforms.ToTensor(),
         transforms.Normalize(3 * [0.5], 3 * [0.5]), ]
    )
    test_transform = transforms.Compose(
        [transforms.Resize((32, 32)),
         transforms.ToTensor(),
         transforms.Normalize(3 * [0.5], 3 * [0.5]),
        ]
    )
    train_ds = datasets.CIFAR10('.', train=True,
                              transform=transform, target_transform=None,
                              download=True)
    test_set = datasets.CIFAR10('.', train=False,
                              transform=test_transform, target_transform=None,
                              download=True)
    # Here we set 'pool_specifics', where we set the transform attribute for the pool.
    active_set = ActiveLearningDataset(train_ds, pool_specifics={
        'transform': test_transform})
    # We start labeling randomly.
    active_set.label_randomly(initial_pool)
    return active_set, test_set
hyperparams = ExperimentConfig()
use_cuda = torch.cuda.is_available()
torch.backends.cudnn.benchmark = True
random.seed(1337)
torch.manual_seed(1337)
if not use_cuda:
    print("warning, the experiments would take ages to run on cpu")

# Get datasets
active_set, test_set = get_datasets(hyperparams.initial_pool)
heuristic = get_heuristic(hyperparams.heuristic)
criterion = CrossEntropyLoss()
model = vgg16(num_classes=10)

if use_cuda:
    model.cuda()
optimizer = optim.SGD(model.parameters(), lr=hyperparams.lr, momentum=0.9)

# Wraps the model into a usable API.
model = EnsembleModelWrapper(model, criterion)

# for prediction we use a smaller batchsize
# since it is slower
active_loop = ActiveLearningLoop(active_set,
                                 model.predict_on_dataset,
                                 heuristic,
                                 hyperparams.query_size,
                                 batch_size=1,
                                 iterations=hyperparams.iterations,
                                 use_cuda=use_cuda)

# We will reset the weights at each active learning step.
init_weights = deepcopy(model.state_dict())

Files already downloaded and verified
Files already downloaded and verified

Presenting EnsembleModelWrapper

EnsembleModelWrapper is similar to ModelWrapper, but instead of training a single model, we will train multiple. Each model will start its training from a different set parameters.

EnsembleModelWrapper methods:

class EnsembleModelWrapper:
    def add_checkpoint(self):
        
        """Add a checkpoint to the list of weights used for inference."""

    def clear_checkpoints(self):
        
(continues on next page)
Clear the list of saved checkpoints.

As you see in the next cell, we call both of these methods alternatively. We train N models, calling add_checkpoint, perform the active learning step and then restart by calling clear_checkpoints.

```python
labelling_progress = active_set._labelled.copy().astype(np.uint16)
report = []
for epoch in tqdm(range(hyperparams.epoch)):
    model.clear_checkpoints()
    # Load the initial weights.
    for model_iter in range(hyperparams.iterations):
        print(f"Training model {model_iter}"),
        model.load_state_dict(init_weights)
        model.model.apply(weights_reset)
        _ = model.train_on_dataset(active_set, optimizer=optimizer, batch_size=hyperparams.batch_size, use_cuda=use_cuda, epoch=hyperparams.training_duration)
        model.add_checkpoint()

    # Get test NLL!
    model.test_on_dataset(test_set, hyperparams.batch_size, use_cuda, average_predictions=hyperparams.iterations)
    metrics = model.metrics

    # We can now label the most uncertain samples according to our heuristic.
    should_continue = active_loop.step()
    # Keep track of progress
    labelling_progress += active_set._labelled.astype(np.uint16)
    if not should_continue:
        break

    test_loss = metrics['test_loss'].value
    logs = {
        "test_nll": test_loss,
        "epoch": epoch,
        "Next Training set size": len(active_set)
    }
    report.append(logs)
```

HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=58.0), HTML(value='')))
1.9 How to use BaaL with Pytorch Lightning

In this notebook we’ll go through an example of how to build a project with Baal and Pytorch Lightning

Useful resources:

- Pytorch Lightning documentation
- Collection of notebooks with other relevant examples

NOTE The API of ActiveLearningMixin and BaalTrainer are subject to change as we are looking for feedback from the community. If you want to help us making this API better, please come to our Gitter or submit an issue.
from torchvision.models import vgg16
from torchvision.transforms import transforms

1.9.1 Model definition

Bellow you can see an example using VGG16

Note the ActiveLearningMixin which we will use to perform active learning. This Mixin expects an active dataset and the following keys in the hparams:

```
iterations: int # How many MC sampling to perform at prediction time.
replicate_in_memory: bool # Whether to perform MC sampling by replicating the batch
```

If you want to modify how the MC sampling is made, you can overwrite predict_step.

```
[ ]: class VGG16(LightningModule, ActiveLearningMixin):
    def __init__(self, **kwargs):
        super().__init__()
        self.save_hyperparameters()
        self.name = "VGG16"
        self.version = "0.0.1"
        self.criterion = CrossEntropyLoss()
        self._build_model()

    def _build_model(self):
        # We use patch_module to swap Dropout modules in the model
        # for our implementation which enables MC-Dropout
        self.vgg16 = patch_module(vgg16(num_classes=self.hparams.num_classes))

    def forward(self, x):
        return self.vgg16(x)

    def training_step(self, batch, batch_idx):
        """
        Lightning calls this inside the training loop
        :param batch:
        :return:
        """
        # forward pass
        x, y = batch
        y_hat = self(x)

        # calculate loss
        loss_val = self.criterion(y_hat, y)

        self.log("train_loss", loss_val, prog_bar=True, on_epoch=True)
        return loss_val

    def test_step(self, batch, batch_idx):
        x, y = batch

```
y_hat = self(x)

# calculate loss
loss_val = self.criterion(y_hat, y)

self.log("test_loss", loss_val, prog_bar=True, on_epoch=True)
return loss_val

def configure_optimizers(self):
    
    return whatever optimizers we want here
    :return: list of optimizers
    
    optimizer = optim.SGD(self.parameters(), lr=self.hparams.learning_rate,
    ...momentum=0.9, weight_decay=5e-4)
    return [optimizer], []

Hyperparameters

[ ]: @dataclass
class HParams:
    batch_size: int = 10
    data_root: str = '/tmp'
    num_classes: int = 10
    learning_rate: float = 0.001
    query_size: int = 100
    iterations: int = 20
    replicate_in_memory: bool = True
    gpus: int = 1

hparams = HParams()

DataModule

We support pl.DataModule, here is how you can define it. By using BaaLDataModule, you do not have to implement pool_dataloader which is the DataLoader that runs on the pool of unlabelled examples.

[ ]: class Cifar10DataModule(BaaLDataModule):
    def __init__(self, data_root, batch_size):
        train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
        ...transforms.ToTensor()])
        test_transform = transforms.Compose([transforms.ToTensor()])
        active_set = ActiveLearningDataset(
            CIFAR10(data_root, train=True, transform=train_transform, download=True),
            pool_specifics={
                'transform': test_transform
            })
        self.test_set = CIFAR10(data_root, train=False, transform=test_transform, ...
        ...download=True)
        super().__init__(active_dataset=active_set, batch_size=batch_size,
Experiment

We now have all the pieces to start our experiment.

1.9.2 Initial labelling

To kickstart active learning, we will randomly select items to be labelled.

```python
[ ]: data_module = Cifar10DataModule(hparams.data_root, hparams.batch_size)
data_module.active_dataset.label_randomly(10)
```

1.9.3 Instantiating BALD

This is used to rank the uncertainty. More info here.

```python
[ ]: heuristic = BALD()
model = VGG16(**asdict(hparams))
```

1.9.4 Create a trainer to generate predictions

Note that we use the BaaLTrainer which inherits the usual Pytorch Lightning Trainer. The BaaLTrainer will take care of the active learning part by performing `predict_on_dataset` on the pool.

```python
[ ]: trainer = BaaLTrainer(dataset=data_module.active_dataset,
heuristic=heuristic,
ndata_to_label=hparams.query_size,
max_epochs=10, default_root_dir=hparams.data_root,
gpus=hparams.gpus,
callbacks=[ResetCallback(copy.deepcopy(model.state_dict()))])
```
1.9.5 Training the model and perform Active learning

Our experiment steps are as follow:

1. Train on the labelled dataset.
2. Evaluate ourselves on a held-out set.
3. Label the top-k most uncertain examples.
4. Go back to 1.

```python
[ ]: AL_STEPS = 100

for al_step in range(AL_STEPS):
    print(f'Step {al_step} Dataset size {len(data_module.active_dataset)}')
    trainer.fit(model, datamodule=data_module) # Train the model on the labelled set.
    trainer.test(model, datamodule=data_module) # Get test performance.
    should_continue = trainer.step(model, datamodule=data_module) # Label the top-k most uncertain examples.
    if not should_continue:
        break
```

1.10 Active Learning for NLP Classification

In this tutorial, we guide you through using our new HuggingFace trainer wrapper to do active learning with transformers models. Any model which could be trained by HuggingFace trainer and has Dropout layers could be used in the same manner.

We will use the SST2 dataset and BertForSequenceClassification as the model for the purpose of this tutorial. As usual, we need to first download the dataset.

Note: This tutorial is intended for advanced users. If you are not familiar with BaaL, please refer to other tutorials.

```
[1]: from datasets import load_dataset
datasets = load_dataset("glue", "sst2", cache_dir="/tmp")
raw_train_set = datasets['train']

Reusing dataset glue (/tmp/glue/sst2/1.0.0/
...7c99657241149a24692c402a5c3f34d4c9f1df5ac2e4c3759fa8a38f6cb29c4)
```

1.10.1 ActiveLearning Dataset

In order to create an active learning dataset, we need to wrap the dataset with baal.ActiveLearningDataset. This requires a torch.utils.Dataset so we propose a baal.active.HuggingFaceDataset that can take a HuggingFace dataset and perform the preprocessing steps.

```
[2]: from baal.active import active_huggingface_dataset
from transformers import BertTokenizer
pretrained_weights = 'bert-base-uncased'

tokenizer = BertTokenizer.from_pretrained(pretrained_model_name_or_path=pretrained_
...weights)
active_set = active_huggingface_dataset(raw_train_set, tokenizer)
```

(continues on next page)
```python
# lets randomly label 100 samples, therefore len(active_set) should be 100
active_set.label_randomly(100)
assert len(active_set) == 100
print(len(active_set.pool))
```

67249

## 1.10.2 Active Learning Model

The process of making a model bayesian is exactly the same as before. In this case, we will get the Bert model and use `baal.bayesian.dropout.patch_module` to make the dropout layer stochastic at inference time.

```python
from copy import deepcopy
import torch
from transformers import BertForSequenceClassification
from baal.bayesian.dropout import patch_module

use_cuda = torch.cuda.is_available()
model = BertForSequenceClassification.from_pretrained(pretrained_model_name_or_path=pretrained_weights)
model = patch_module(model)
if use_cuda:
    model.cuda()
init_weights = deepcopy(model.state_dict())
```

Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertForSequenceClassification: 
- This IS expected if you are initializing BertForSequenceClassification from the checkpoint of a model trained on another task or with another architecture (e.g., initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertForSequenceClassification from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model). Some weights of BertForSequenceClassification were not initialized from the model.
- `classifier.weight`, `classifier.bias`
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
1.10.3 Heuristic

As already implemented and useful in all classification cases, we continue using BALD as our active learning heuristic. Note: ActiveLearning for NLP tasks is an open and challenging field and hence, designing a proper heuristic is out of the scope of this tutorial. We encourage any pull request that would propose better heuristics.

```python
from baal.active import get_heuristic

heuristic = get_heuristic('bald')
```

1.10.4 HuggingFace Trainer Wrapper

If you are not familiar with the HuggingFace trainer module please start here. HuggingFace Trainer is one of the most popular libraries to train Transformer models. In order to do active learning, we need the prediction to be run over every sample in pool for number of iterations and hence our wrapper `baal.BaalTransformersTrainer` will provide this functionality on top of the provided functionalities in the Trainer module. In the rest of this tutorial, we show how to initialize the `baal.active.active_loop.ActiveLearningLoop` and how to do Active Training.

```python
from transformers import TrainingArguments
from baal.transformers_trainer_wrapper import BaalTransformersTrainer
from baal.active.active_loop import ActiveLearningLoop

# Initialization for the huggingface trainer
training_args = TrainingArguments(
    output_dir= '.',  # output directory
    num_train_epochs=5,  # total # of training epochs per AL step
    per_device_train_batch_size=16,  # batch size per device during training
    per_device_eval_batch_size=64,  # batch size for evaluation
    weight_decay=0.01,  # strength of weight decay
    logging_dir= '.',  # directory for storing logs
)

# create the trainer through Baal Wrapper
baal_trainer = BaalTransformersTrainer(model=model,
    args=training_args,
    train_dataset=active_set,
    tokenizer=None)

active_loop = ActiveLearningLoop(active_set,
    baal_trainer.predict_on_dataset,
    heuristic, 10, iterations=3)

for epoch in range(2):
    baal_trainer.train()

    should_continue = active_loop.step()

    # We reset the model weights to relearn from the new train set.
    baal_trainer.load_state_dict(init_weights)
    baal_trainer.lr_scheduler = None
    if not should_continue:
        # (continues on next page)
```
# at each Active step we add 10 samples to labelled data. At this point we should have
→ 30 samples added
# to the labelled part of training set.
print(len(active_set))

1.11 How to use BaaL with Scikit-Learn models

In this tutorial, you will learn how to use BaaL on a scikit-learn model. In this case, we will use RandomForestClassifier.

This tutorial is based on the tutorial from Saimadhu Polamuri.

First, if you have not done it yet, let’s install BaaL.

```bash
pip install baal
```

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split
```
```python
import pandas as pd
dataset = pd.read_csv(data)
dataset.columns = HEADERS

# Handle missing labels
dataset = dataset[dataset[HEADERS[6]] != '?']

# Split
train_x, test_x, train_y, test_y = train_test_split(dataset[HEADERS[1:-1]],
                                                     dataset[HEADERS[-1]],
                                                     train_size=0.7)

clf = RandomForestClassifier()
clf.fit(train_x, train_y)

# Get metrics
predictions = clf.predict(test_x)
print("Train Accuracy :: ", accuracy_score(train_y, clf.predict(train_x)))
print("Test Accuracy :: ", accuracy_score(test_y, predictions))
print(" Confusion matrix ", confusion_matrix(test_y, predictions))
```

Now that you have a trained model, you can use it to perform uncertainty estimation. The SKLearn API directly proposes `RandomForestClassifier.predict_proba` which would return the mean response from the RandomForest.

But if you wish to try one of our heuristics in `baal.active.heuristics`, here's how.

```python
import numpy as np
from baal.active.heuristics import BALD
print(f"Using {len(clf.estimators_)} estimators")

# Predict independently for all estimators.
x = np.array(list(map(lambda e: e.predict_proba(test_x), clf.estimators_)))
```

(continues on next page)
# Roll axis because BaaL expect [n_samples, n_classes, ..., n_estimations]
x = np.rollaxis(x, 0, 3)
print("Uncertainty per sample")
print(BALD().compute_score(x))

print("Ranks")
print(BALD()(x))

Using 10 estimators
Uncertainty per sample

Using 10 estimators
Uncertainty per sample

Ranks

1.11. How to use BaaL with Scikit-Learn models
1.11.1 Active learning with SkLearn

You can also try Active learning by using `ActiveNumpyArray`.

**NOTE**: Because we focus on images, we have not made experiments on this setup.

```python
[14]: from baal.active.dataset import ActiveNumpyArray
dataset = ActiveNumpyArray((train_x, train_y))

# We start with a 10 labelled samples.
dataset.label_randomly(10)
heuristic = BALD()

# We will use a RandomForest in this case.
clf = RandomForestClassifier()
def predict(test, clf):
    # Predict with all fitted estimators.
    x = np.array(list(map(lambda e: e.predict_proba(test[0]), clf.estimators_)))

    # Roll axis because BaaL expect [n_samples, n_classes, ..., n_estimations]
    x = np.rollaxis(x, 0, 3)
    return x

for _ in range(5):
    print("Dataset size", len(dataset))
    clf.fit(*dataset.dataset)
    predictions = clf.predict(test_x)
    print("Test Accuracy :: ", accuracy_score(test_y, predictions))
    probs = predict(dataset.pool, clf)
    to_label = heuristic(probs)
    ndata_to_label = 10
    if len(to_label) > 0:
        dataset.label(to_label[: ndata_to_label])
    else:
        break

Dataset size 10
Test Accuracy :: 0.9219512195121952
Dataset size 20
Test Accuracy :: 0.9658536585365853
Dataset size 30
Test Accuracy :: 0.9414634163414634
Dataset size 40
```

(continues on next page)
1.12 Report on using Dirichlet calibration for active learning

A paper recently published at NeurIPS 2019 proposes to estimate a dirichlet distribution to model the uncertainty of a prediction.

To achieve that, they add a new linear layer at the end of the network and train it individually on a held-out set. Here is a figure from the authors’ NeurIPS 2019 presentation. You can find the full presentation on the website above.

Our hypothesis is as follows: by modelling the uncertainty on an held-out set, we want to create a better estimation of the overall uncertainty.

A common flaw on uncertainty estimation technique using Variational Inference (VI) such as MC-Dropout is that they only model the epistemic uncertainty ie. the uncertainty of the model.

Current SotA methods for active learning rely on VI to estimate the model uncertainty. Due to the lack of data, those methods are often not better than random due to overfitting or underfitting. By introducing calibration, we hope to reduce the need for VI and to rely on calibrated predictions to select the next sample to label.

1.12.1 Methodology

Our methodology follows a standard active learning pipeline, but we add a new training set: $D_{L}$ which is used to train the calibration layer. After training the model on the training set $D_{\text{train}}$ to convergence, we train it on this held-out set and train the newly added layer.

We call the augmented model $M_{\text{calib}}$. We perform the sample selection using one of the following techniques:

- Entropy: $\sum_c p_{i} \log(p_{i})$
- BALD using MC-Dropout: $H[y \mid x, D_{\{L\}}] - E_{p(w \mid D_{L})}(H[y \mid x, w])$
• Uniform random selection

Because we want to analyze the actual gain of using calibration, we compare the effect of using $M$ versus $M_{calib}$ across all techniques.

1.12.2 Experiments

We test our hypothesis on CIFAR10 using a VGG-16. We initially label 1000 samples and label 100 samples per step. When using MC-Dropout, we set the number of Monte-Carlo sampling to 20.

Testing calibration

We first want to ensure that calibration works properly. In Fig. 2, we show that throughout the active learning procedure, the calibrated loss is better than the non-calibrated loss.

![Comparison between the calibrated loss and the uncalibrated loss.](image1)

**Fig. 1:** Comparison between the calibrated loss and the uncalibrated loss.

Furthermore, we compute the ECE between both cases.

![Comparison between ECE for both Calibrated BALD and BALD.](image2)

**Fig. 2:** Comparison between ECE for both Calibrated BALD and BALD.
Impact of calibration on active learning

For each method, we present the calibrated NLL at each active learning step. We want to compare the selection process between $M$ and $M_{calib}$. Our reasoning is as follow. We want to see if the calibrated model would pick better items over the normal one. To do so we make two experiments, one where we use $M$ to select the new samples and the other uses $M_{calib}$. In both cases, we will get a calibrated model to compare the calibrated loss.

**Fig. 3:** Comparison between a calibrated selector and an uncalibrated one using BALD.

**Fig. 4:** Comparison between a calibrated selector and an uncalibrated one using Entropy.

In addition, we show that BALD is still better in all cases.

### 1.12.3 Discussion

While we have not seen improvements by using calibration on an active learning benchmark, we still find this report useful. Active learning is but a part of the Human-ai-interaction (HAII) process. By adding an easy to use calibration method, we can further the collaboration between the human and our model.

By giving more nuanced predictions, the model is deemed more trustable by the human annotator.
Calibration in BaaL

With BaaL 1.2, we add a new module based on this report. We propose new tools and methods to calibrate your model. Our first method will be a Pytorch implementation of the Dirichlet Calibration method. Here is an example:

```python
from baal.calibration import DirichletCalibrator
from baal.modelwrapper import ModelWrapper

""
Get your train and validation set. In addition, you need a
held-out set to calibrate your model.
""

train_ds, calib_ds, valid_ds = get_datasets()
wrapper = ModelWrapper(MyModel(), criterion=YourCriterion())

# Make a calibrator object.
calibrator = DirichletCalibrator(wrapper, 2, lr=0.001, reg_factor=0.001)

# Train your model as usual.
wrapper.train_on_dataset(train_ds, SGD(...), batch_size=32, epoch=30, use_cuda=True)

# Calibrate your model on a held-out set.
calibrator.calibrate(calib_ds, valid_ds, batch_size=10, epoch=5,
                     use_cuda=True,
                     double_fit=True, workers=4)
calibrated_model = calibrator.calibrated_model
```

Fig. 5: Comparison between calibrated selectors.
1.12.4 Conclusion

In this report, we tested the effect of calibration on an active learning pipeline. In addition, our code is freely available in our open source library.

1.12.5 Author(s)

- Frédéric Branchaud-Charron
- Parmida Atighehchian

1.13 Report on the case of double descent in active learning

Double descent was first investigated by OpenAI in 2019. They have also published their hypothesis on why double descent happens in some training tasks, in this paper.

During our first rounds of experiments with MCDropout for active learning, we came around double descent incident in several cases and hence, it was all the reason to figure out why and when it happens. Double descent is harmful to the active learning procedure as it is a sign that the model did not learn the current labelled set properly and therefore the next set of uncertainties would not be valid.

A combination of several conditions and hyper-parameters could result in a double descent. Below, we will define each condition/hyper-parameter, and then we describe what combinations would result in double descent occurrence. We will end this blog post with our hypothesis as to why double descent happens in each case.

1.13.1 Training Condition

Early Stopping

While early stopping helps prevent overfitting, experiments show that in the case of active learning early stopping along with other conditions would result in the negative effect of double descent. In our experiments, the trials which are tested using early stopping are marked by $\text{lbw}_{\text{Y}}$ and $p$ will define the number of patience epochs to make sure that the loss is not oscillating.

Resetting the weights of the model

During active learning, we reset the model weights to some initial/random weights after each active learning step, so that the model is ready to learn from the new distribution of labelled samples without the prior bias of the previous distribution of labelled samples. In this paper, it is shown that the last few layers have the most influential weights for estimating model uncertainty in Bayesian neural architecture, and hence, we have made it possible in our pipeline to do active learning with only resetting the weights for the last few layers of a network. In this study, you will see if taken proper measurements resetting only the linear layers could be enough for removing this prior bias, and hence, it would speed up the process by not forcing the model to learn from scratch. However, depending on whether we are using early stopping or not this can introduce double descent and be harmful to the procedure. In our experiments $\text{rs: full}$ defines, resetting the model weights to their initial weights and $\text{rs: partial}$ indicates resetting the weights of the linear layers only.
1.13.2 Hyper-parameters

Weight decay

The different set of experiments are tested also with different weight decays and in some cases, changing the weight decay could prevent double descent. Weight decay parameter value is marked as $wd$.

1.13.3 Experiments and analysis

We ran 4 categories of experiments: Dataset: CIFAR10 Model: Vgg16 trained on imagenet

Fig. 6: Using early stopping and reset the weights of the linear layers after each active learning step.

Fig. 7: Using early stopping and reset all the weights after each active learning step.

In the first two experiments, if we are using early stopping, the partial reset will provoke a double descent. A closer look in the second diagram shows that although in the case of fully resetting the model weights, we can prevent the double descent phenomenon, it is highly dependent on the number of patience epochs $p$. In the case of early stopping, one shall give the model enough time to learn and stabilize before stopping the training in each active learning step.

Moving to the last two experiments, we show that not using early stopping results in smoother active training. The first graph shows us that using partial resetting could be very tricky for active learning. To prevent double descent and hence a smooth training with partial resetting, proper tuning the model using regularization methods such as `weight_decay` is a necessity. In our case, if we don’t use weight decay, the double descent still will happen although with a negligible peak. Moreover, letting the model train well before performing uncertainty estimation is another key to encourage
Fig. 8: Overfitting the training set and reset the weights of the linear layers after each active learning step.

Fig. 9: Overfitting the training set and reset all the weights after each active learning step.
smooth training, we show the difference between letting the model to train for 10 epochs vs 5 epochs before adding samples to the labelled set.

NOTE: In the case of not using early stopping, \( p \) is used to show the number of epochs we train the model before estimating uncertainties and increase the labelled set. All in all, not using early stopping and fully resetting the model weights i.e. the last graph, could certify a smooth training procedure without being worried about other elements such as weight decay.

1.13.4 Our Hypothesis

After analysing the results of our experiments, we tend to agree with OpenAI on the hypothesis as to why double descent might happen. However, the case of active learning is more complex as the dataset gets more and more complex after each active learning step. Even though we do not change the model architecture, we do reset the model weights, so that the model can fit the new dataset again from scratch.

On the other hand, active learning is an expensive procedure and sometimes we need to minimize this cost by resetting only parts of the model weights. This of course would lead to the model having difficulty to learn as we grow the dataset, and as OpenAI reports, “adding more data is harmful to the training”. However, we hypothesise that it is still possible to minimize the cost of training using partial resetting if one uses early stopping and closely track the model training to tune the weight decay properly and hence lead the model to smoothly get adjusted to the new dataset at each active learning step. We argue that this is the best way of getting a competing loss while reducing the cost of active learning.

1.13.5 Author(s)

- Parmida Atighehchian
- Frédéric Branchaud-Charron

1.13.6 Note to our readers

As we are excited to share our experiments and hypotheses with you, we do not claim that this report is satisfactory. Therefore, we would encourage our BaaL users to help us spread informations and results on this particular problem. Please contact us if you make further discovery on this important question.

1.14 Can active learning preemptively mitigate fairness issues?

*By Parmida Atighehchian*

The purpose of this notebook is to demonstrate the preliminary results of our recent contribution to ICLR workshop of Responsible AI 2021. We show that active learning could help in creating fairer datasets without the need to know the bias in the dataset. This is important since in real scenarios, the source of bias is often unknown. Using active learning (i.e. BALD), we show that the prior knowledge of the bias is not necessary and hence it could be easier to integrate this setup in pipelines to make sure that the dataset is generally fairer and the possible biases are reduced.

For the purpose of this demo, we use Synbols dataset. Synbols is the new state of the art generating synthetic datasets. The Dockerfile is located at `baal/notebooks/fairness/Docker_biased_data`.

More resources on BaaL:

- Literature review
- Active learning dataset and training loop classes
• Methods for approximating bayesian posteriors
• Full active learning example

If you have any question, please submit an issue or reach out on Gitter.

1.14.1 Introducing bias in dataset

Using Synbols, we will generate a character classification dataset with an important correlation between the character and the color. There is a correlation between the color blue and the character a:

\[ p(\text{char} = \text{a} | \text{color} = \text{blue}) = 90\% \]

and there is a correlation between the color red and the character d:

\[ p(\text{char} = \text{d} | \text{color} = \text{red}) = 90\% \]

[1]:
```python
import numpy as np
from math import pi
from synbols.data_io import pack_dataset
from synbols import drawing
from synbols import generate

class InfoSolid(drawing.SolidColor):
    def attribute_dict(self):
        d = super().attribute_dict()
        d['color'] = self.color
        return d

rng = np.random.RandomState(1337)
p = .1
blue = (0,0,255)
red = (255, 0, 0)

class SpuriousSampler:
    def __init__(self, p):
        self.p = p

    def __call__(self, seed):
        """Makes color dependant on symbol.""
        rng = np.random.RandomState(seed)
        color = [blue, red][rng.choice([0, 1], p=[self.p, 1-self.p])]
        char = rng.choice(['a', 'd'])
        color_p = {'a':self.p, 'd':1-self.p}[char]
        color = [blue, red][rng.choice([0, 1], p=[color_p, 1-color_p])]
        fg = InfoSolid(color)
        fg.color = color

        attr_sampler = generate.basic_attribute_sampler(
            char=char, foreground=fg, background=None, inverse_color=False,
            resolution=(64, 64))
        d = attr_sampler()
        return d
```

(continues on next page)
```python
def make_dataset(p, seed, num):
    attribute_sampler = SpuriousSampler(p=p)
    x, mask, y = pack_dataset(generate.dataset_generator(attribute_sampler, num,
        generate.flatten_mask, dataset_seed=seed))
    for yi in y:
        yi['color'] = 'red' if yi['foreground']['color'] == 'red' else 'blue'
    return (x,y,y)
```

```python
train_set = make_dataset(p=0.9, seed=1000, num=10000)
test_set = make_dataset(p=0.5, seed=2000, num=5000)
dataset = {'train': train_set, 'test': test_set}
```

### 1.14.2 Prepare model and dataset to be used in BaaL setup

As usual we wrap the `train_set` in `ActiveLearningDataset` and using vgg16 as default model, we use the BaaL's `patch_module` to create a dropout layer which performs in inference time.

```python
def get_datasets(dataset : Dict, initial_pool: int, attribute:str, target_key:str):
    """
    Get the dataset for the experiment.
    Args:
        dataset: The symbol generated dataset.
        initial_pool: Initial number of items to label.
        attribute: Key where the sensitive attribute is.
        target_key: Key where the target is.
    Returns:
        ActiveLearningDataset with `initial_pool` items labelled
    Test dataset
    """
    transform = transforms.Compose(
        [transforms.ToPILImage(),
         transforms.Resize((IMG_SIZE, IMG_SIZE)),
         transforms.RandomHorizontalFlip(),
         transforms.RandomRotation(30),
         transforms.ToTensor(),
         transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])
    test_transform = transforms.Compose([transforms.ToPILImage(),
                                          transforms.Resize((IMG_SIZE, IMG_SIZE)),
                                          transforms.ToTensor(),
                                          transforms.Normalize((0.4914, 0.4822, 0.4465),
```

(continues on next page)
train_ds = dataset['train']
test_ds = dataset['test']
ds = SynbolDataset(*train_ds, target_key=target_key, attribute=attribute,
                  transform=transform)

test_set = SynbolDataset(*test_ds, target_key=target_key, attribute=attribute,
                         transform=test_transform)

active_set = ActiveLearningDataset(ds, pool_specifics={'transform': test_transform})
active_set.label_randomly(initial_pool)
return active_set, test_set

[4]: from torchvision import models
    from torch.hub import load_state_dict_from_url
    from baal.bayesian.dropout import patch_module

    #set use_cuda to False if you don't have access to GPUS
    use_cuda=True

    model = models.vgg16(pretrained=False, num_classes=2)
    weights = load_state_dict_from_url('https://download.pytorch.org/models/vgg16-397923af.pth')
    weights = {k: v for k, v in weights.items() if 'classifier.6' not in k}
    model.load_state_dict(weights, strict=False)

    # change dropout layer to MCDropout
    model = patch_module(model)

    if use_cuda:
        model.cuda()

We wrap the pytorch criterion to accommodate target being a dictionary.

[5]: from torch import nn

class Criterion(nn.Module):
    def __init__(self, crit):
        super().__init__()
        self.crit = crit

    def forward(self, input, target):
        return self.crit(input, target['target'])
Training

Let’s now train the model with active learning. As usual, we compare bald with random but this time, we are looking for something else in the results!

```python
from copy import deepcopy
from tqdm import tqdm
import pandas as pd
import torch
from torch import optim
from torch.nn import CrossEntropyLoss
from baal.modelwrapper import ModelWrapper
from baal.active.heuristics import BALD
from baal.active.active_loop import ActiveLearningLoop
from active_fairness.metrics import FairnessMetric
import sklearn.metrics as skm

heuristics = ['bald', 'random']

logs = {'bald': {}, 'random': {}}

for heuristic_name in heuristics:
    active_set, test_set = get_datasets(dataset, initial_pool=500, attribute='color',
                                        target_key='char')

    heuristic = get_heuristic(name=heuristic_name, shuffle_prop=0.0)

    criterion = Criterion(CrossEntropyLoss())

    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9, weight_decay=5e-4)

    wrapped_model = ModelWrapper(model, criterion)

    wrapped_model.add_metric('aggregate_res', lambda: FairnessMetric(skm.accuracy_score,
                                                                       name='acc',
                                                                       attribute='color'))

    # save imagenet weights
    init_weights = deepcopy(model.state_dict())

    bald = BALD()

    # for prediction we use a smaller batchsize
    # since it is slower
    active_loop = ActiveLearningLoop(active_set, wrapped_model.predict_on_dataset,
                                      heuristic, 50,
                                      batch_size=16,
                                      iterations=20,
```

(continues on next page)
```
use_cuda=use_cuda,
workers=0)

learning_epoch = 20
for epoch in tqdm(range(100000)):
    wrapped_model.load_state_dict(init_weights)
    wrapped_model.train_on_dataset(active_set, optimizer, batch_size=32,
                                   epoch=learning_epoch, use_cuda=True, workers=12)

    # Validation!
    wrapped_model.test_on_dataset(test_set, batch_size=32, use_cuda=use_cuda,
                                   workers=12, average_predictions=20)

    should_continue = active_loop.step()
    if not should_continue:
        break

    # Send logs
    fair_train = wrapped_model.metrics['train_aggregate_res'].value
    epoch_logs = {
                  'epoch': epoch,
                  'test_loss': wrapped_model.metrics['test_loss'].value,
                  'active_train_size': len(active_set)}

    agg_res = {'train_' + k: v for k, v in fair_train.items()}
    epoch_logs.update(agg_res)
    for k, v in epoch_logs.items():
        if k in logs[heuristic_name].keys():
            logs[heuristic_name][k].append(v)
        else:
            logs[heuristic_name][k] = [v]

    if len(active_set) > 2000:
        break
```

1.14.3 Results and Discussion

Below we show the number of samples added to each subcategory (i.e. character with a specific color) as the training goes on. Interesting result is that the number of samples added to the minority group of each character increases using bald where as random picks samples in a random setup and hence having more samples given a protected attribute (here color), random has more samples of a certain color to pick. This indicates that active learning with bald generally leads to a more fair dataset.

```
import matplotlib.pyplot as plt
%matplotlib inline

x = logs['bald']['epoch']
fig, ((ax0, ax1), (ax2, ax3)) = plt.subplots(nrows=2, ncols=2, sharex=True,
                                          figsize=(12, 6))
plots_target = [('minority count for character a', 'train_count_0_red'),
                ('minority count for character b', 'train_count_1_red'),
                ('minority count for character c', 'train_count_2_red')]
```

1.14. Can active learning preemptively mitigate fairness issues?
We demonstrate the test_loss and training_size using bald vs random as heuristics. As it is shown, the training size increases with the same pace but the above graphs shows the underlying difference in the existing samples for each class which then results in also a better loss decrease using bald.

```
[16]: x = logs['bald']['epoch']
fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, sharex=True, figsize=(12, 6))
ax0.set_title('training size')
ax0.plot(x, logs['bald']['active_train_size'], color='r', label='BALD')
ax0.plot(x, logs['random']['active_train_size'], color='b', label='Uniform')

ax1.set_title('test loss')
ax1.plot(x, logs['bald']['test_loss'], color='r', label='BALD')
ax1.plot(x, logs['random']['test_loss'], color='b', label='Uniform')
ax1.legend()
fig.show()
```
1.15 Active learning literature

This page is here to collect summaries of papers that focus on active learning. The idea is to share knowledge on recent developments in active learning.

If you’ve read a paper recently, write a little summary in markdown, put it in the folder docs/literature and make a pull request. You can even do all of that right in the github web UI!

1.15.1 The theory behind Bayesian active learning

In this document, we keep a list of the papers to get you started in Bayesian deep learning and Bayesian active learning. We hope to include a summary for each of them in the future, but for now we have this list with some notes.

**How to estimate uncertainty in Deep Learning networks**

- Excellent tutorial from AGW on Bayesian Deep Learning
  - This is inspired by his publication Bayesian Deep Learning and a Probabilistic Perspective of Generalization
- Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning (Gal and Ghahramani, 2016)
  - This describes Monte-Carlo Dropout, a way to estimate uncertainty through stochastic dropout at test time
- Bayesian Uncertainty Estimation for Batch Normalized Deep Networks (Teye et al. 2018)
  - This describes Monte-Carlo BatchNorm, a way to estimate uncertainty through random batch norm parameters at test time
- Bayesian Deep Learning and a Probabilistic Perspective of Generalization (Gordon Wilson and Izmailov, 2020)
  - Presentation of multi-SWAG a mix between VI and Ensembles.
Advances in Variational inference (Zhang et al, 2018)
  – Gives a quick introduction to VI and the most recent advances.

A Simple Baseline for Bayesian Uncertainty in Deep Learning (Maddox et al. 2019)
  – Presents SWAG, an easy way to create ensembles.

Bayesian active learning

Deep Bayesian Active Learning with Image Data (Gal and Islam and Ghahramani, 2017)
  – Fundamental paper on how to do Bayesian active learning. A must read.

Sampling bias in active learning (Dasgupta 2009)
  – Presents sampling bias and how to solve it by combining heuristics and random selection.

Bayesian Active Learning for Classification and Preference Learning (Houlsby et al. 2011)
  – Fundamental paper on one of the main heuristic BALD.

Bayesian active learning on NLP

Deep Bayesian Active Learning for Natural Language Processing: Results of a Large-Scale Empirical Study (Siddhant and Lipton, 2018)
  – Experimental paper on how to use Bayesian active learning on NLP tasks.

1.15.2 Additional papers that are interesting

In this section, we put additional papers that can be interesting.

Deep InfoMax

URL

The authors propose a new method for unsupervised representation learning by using adversarial networks.

As shown in Fig. 3, the features extracted from the local features (top of the network) and global features (the embedding).

Figure 3: Maximizing mutual information between local features and global features. First we encode the image to a feature map that reflects some structural aspect of the data, e.g. spatial locality, and we further summarize this feature map into a global feature vector (see Figure 1). We then concatenate this feature vector with the lower-level feature map at every location. A score is produced for each local-global pair through an additional function (see the Appendix A.2 for details).
A key component of their method is that they use three discriminators.

- Global discriminator
  - Compare the global encoding to the local feature.
- Local discriminator
  - Compare the local features.
- Prior discriminator
  - Force the embedding to a known prior.

They achieve impressive results on classification and on other tasks. For example, by using a SVM on the embeddings, they get an accuracy of 75.21% on CIFAR10 and 49.74% on CIFAR100. For comparison, training the same generator in a fully-supervised setting would get 75.39% on CIFAR10 and 42.27% on CIFAR100.

Uncertainty Estimation Using a Single Deep Deterministic Neural Network

Authors: Joost van Amersfoort, Lewis Smith, Yee Whye Teh and Yarin Gal


Problematic

Using MC-Dropout or ensembles are expensive to run. In this paper, the authors propose to add a gradient penalty to reliably detect out-of-distribution data. In addition, the inference phase is fast because we can perform the uncertainty estimation in a single pass.

Deterministic Uncertainty Quantification (DUQ)

DUQ uses a RBF Network to compute centroids for each class. The model is trained by minimizing the number of false assignment between a class centroid and its members.

Loss function

For a model f, a centroid matrix W and a centroid e, we compute the similarity using a RBF kernel. Theta is a hyper parameter.

\[ K_c(f_\theta, e_c) = \exp(-\frac{1}{2\sigma^2}||W_c f_\theta(x) - e_c||_2^2) \]

with this similarity we can make a prediction by selecting the centroid with the highest similarity.

The loss function is now simply

\[ L(x, y) = -\sum_c y_c \log(K_c) + (1 - y_c) \log(1 - K_c), \]

where \( K_c(f_\theta, e_c) = K_c \)

After each batch, we update the centroid matrix using an exponential moving average.
**Regularization**

To avoid feature collapse, the authors introduce a gradient penalty directly applied to $K_c$: $\lambda * (\| \nabla_x \sum_c K_c \|^2_2 - 1)^2$ where 1 is the Lipschitz constant. In their experiments, they use $\lambda = 0.05$.

In summary, this simple technique is faster and better than ensembles. It also shows that RBF networks work on large datasets.

**Gaussian YOLOV3**

This paper proposes on how to change the deterministic bounding box output of object detection models to uncertainty score. The only proposed change in the model is to replace the detection layer with a single Gaussian model. Hence, the model will output the mean and variance of each coordinated (a vector of size 8 instead of 4).

![Image of Gaussian parameter of bounding box coordinates](image)

**Figure 2:** Components in the prediction box of proposed algorithm.

The loss function of the bounding box detection section would be changed to negative log likelihood as below:

$$L_{\mu_x} = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(x_{ijk}^{G|\mu_x}(x_{ijk}), \Sigma_x(x_{ijk}))) + \varepsilon)$$

$$L_{\mu_y} = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(y_{ijk}^{G|\mu_y}(y_{ijk}), \Sigma_y(y_{ijk}))) + \varepsilon)$$

$$L_{\mu_w} = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(w_{ijk}^{G|\mu_w}(w_{ijk}), \Sigma_w(w_{ijk}))) + \varepsilon)$$

$$L_{\mu_h} = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(h_{ijk}^{G|\mu_h}(h_{ijk}), \Sigma_h(h_{ijk}))) + \varepsilon),$$

$$L_y = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(y_{ijk}^{G|\mu_y}(y_{ijk}), \Sigma_y(y_{ijk}))) + \varepsilon)$$

$$L_w = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(w_{ijk}^{G|\mu_w}(w_{ijk}), \Sigma_w(w_{ijk}))) + \varepsilon)$$

$$L_h = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(h_{ijk}^{G|\mu_h}(h_{ijk}), \Sigma_h(h_{ijk}))) + \varepsilon),$$

$$L_{\mu_x} = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(x_{ijk}^{G|\mu_x}(x_{ijk}), \Sigma_x(x_{ijk}))) + \varepsilon)$$

$$L_{\mu_y} = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(y_{ijk}^{G|\mu_y}(y_{ijk}), \Sigma_y(y_{ijk}))) + \varepsilon)$$

$$L_{\mu_w} = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(w_{ijk}^{G|\mu_w}(w_{ijk}), \Sigma_w(w_{ijk}))) + \varepsilon)$$

$$L_{\mu_h} = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(h_{ijk}^{G|\mu_h}(h_{ijk}), \Sigma_h(h_{ijk}))) + \varepsilon),$$

$$L_y = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(y_{ijk}^{G|\mu_y}(y_{ijk}), \Sigma_y(y_{ijk}))) + \varepsilon)$$

$$L_w = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(w_{ijk}^{G|\mu_w}(w_{ijk}), \Sigma_w(w_{ijk}))) + \varepsilon)$$

$$L_h = - \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \gamma_{ijk} \log(\mathcal{N}(h_{ijk}^{G|\mu_h}(h_{ijk}), \Sigma_h(h_{ijk}))) + \varepsilon),$$
Results

The main concern of using Gaussian approaches is their memory consumption, however as they only propose using a single Gaussian model for the detection layer, the memory consumption is almost not affected whereas mAP is increased by 3% on the two datasets of their choice (KITTI, BDD) comparing to YOLOV3. Based on the provided results, their proposed model increases the TP on the both datasets at least by 4% and reduces the FP by at least 40%.

Scalable k-Means Clustering via Lightweight Coresets

Note This review will focus on the Coreset approach and not so much on the k-means.

Paper pdf

This paper presents a novel Coreset algorithm called Light Coreset.

Let $X$ be the dataset, $d$ a distance function and $\mu(X)$ the mean of the dataset per feature.

We compute the distribution $q$ with:

$$q(x) = 0.5 * \frac{1}{|X|} + 0.5 * \frac{d(x, \mu(X))^2}{\sum_{x' \in X} d(x', \mu(X))^2}, \text{ where } x \in X.$$ 

We can then select $m$ samples by sampling from this distribution. For their experiments, they used the L2 distance for $d$.

Let $A$ be the first part of the equation $q$ and $B$ the second. The authors offer the following explanation:

The first component (A) is the uniform distribution and ensures that all points are sampled with nonzero probability. The second component (B) samples points proportionally to their squared distance to the mean of the data. The intuition is that the points that are far from the mean of the data have a potentially large impact on the quantization error of a clustering. The component (B) ensures that these potentially important points are sampled frequently enough.
Advantages

• Their method is by order of magnitude faster than standard coresets and when tested on some UCI datasets, they perform better as well.

Bayesian Batch Active Learning as Sparse Subset Approximation

Robert Pinsler, Jonathan Gordon, Eric Nalisnick and José Miguel Hernández-Lobato

Published at NeurIPS 2019

ArXiv

Code

A known issue of BALD, when used in Batch Active Learning is that it selects highly correlated samples. By combining BNNs with a novel coreset algorithm, the authors propose a way to estimate the true posterior data distribution.

In brief, they want to select a batch $D'$ such that the posterior distribution best approximate the complete data posterior. Because we do not know the complete posterior, the authors approximate it using the predictive distribution. The idea is summarized in Eq. 4.

$$
\mathbb{E}_{\mathcal{Y}_p} \left[ \log p(\theta | D_0 \cup (\mathcal{X}_p, \mathcal{Y}_p)) \right] = \mathbb{E}_{\mathcal{Y}_p} \left[ \log p(\theta | D_0) + \log p(\mathcal{Y}_p | \mathcal{X}_p, \theta) - \log p(\mathcal{Y}_p | \mathcal{X}_p, D_0) \right] \\
= \log p(\theta | D_0) + \mathbb{E}_{\mathcal{Y}_p} \left[ \log p(\mathcal{Y}_p | \mathcal{X}_p, \theta) + \mathbb{H}[\mathcal{Y}_p | \mathcal{X}_p, D_0] \right] \\
= \log p(\theta | D_0) + \sum_{m=1}^{M} \left( \mathbb{E}_{\mathcal{Y}_m} \left[ \log p(\mathcal{Y}_m | \mathcal{x}_m, \theta) \right] + \mathbb{H}[\mathcal{Y}_m | \mathcal{x}_m, D_0] \right),
$$

This measure can be optimized using Frank-Wolfe which uses the dot-product $\langle L_m, L_n \rangle$ to estimate the affectations.

While a closed-form procedure exists to compute this dot-product, it is expensive to run ($O(||P||^2)$). The authors suggest the use of random projections drawn from the parameters distribution $\hat{\pi}$. This approximation makes the algorithm $O(||P||J)$, where $J$ is the number of samples drawn from $\hat{\pi}$.

In Fig 4, the authors show results on multiple datasets showing that their method is better than BALD or Entropy.

![Figure 4: Test accuracy on classification tasks over 5 seeds. Error bars denote two standard errors.](image)
My two cents

This method is interesting as it is less computationally expensive than BatchBALD with the same goal in mind i.e. solving the correlated selection in BALD.

In their work, they use a batch size of 3000. From our experiments, we know that BALD does work with a lower, but still large query size (250 for example). While this shows that their method works with high batch size, I would have liked to see a comparison between their method and BALD on a smaller query size. In Annex D, they show that MC-Dropout is also affected by the large query size.

Variational adversarial Active Learning (VAAL)

This paper proposes a new Active Learning algorithm that is not task-dependent. To do so, they use a GAN architecture where the generator is a beta-VAE and the discriminator is a simple MLP. This MLP takes the embedding and predicts whether or not the input is a labeled sample.

![VAAL Diagram]

Figure 1. Our model learns the distribution of labeled data in a latent space using a VAE optimized using both reconstruction and adversarial losses. A binary classifier predicts unlabeled examples and sends them to an oracle for annotations. The VAE is trained to fool the adversarial network to believe that all the examples are from the labeled data while the adversarial classifier is trained to differentiate labeled from unlabeled samples.

The loss is then the sum of the VAE Loss and the adversarial loss.

The next sample to label is chosen by the discriminator. From all the samples from the pool, the sample with the highest certainty is chosen.
Results

They present impressive results on classification and segmentation. We should note that in their results, MC-Dropout is the worst method. This is not what we've been seeing in practice.

![Graphs showing performance on various datasets](image)

Figure 2. VAAL performance on classification tasks using CIFAR10, CIFAR100, Caltech-256, and ImageNet compared to Core-set [40], Ensembles w. VarR [1], MC-Dropout [13], DBAL [14], and Random Sampling.

Efficiency

Their method is much faster than other methods such as MC-Dropout. During inference, you only need to run the encoder and the discriminator once. On CIFAR10, they are 8x faster than MC-Dropout.
INDICES AND TABLES

- genindex
- search
predict_on_dataset_generator()  
(baal.ModelWrapper method), 13
predict_on_dataset_generator()  
(baal.utils.pytorch_lightning.BaalTrainer method), 22

R
Random (class in baal.active.heuristics), 21
reorder_indices() (baal.active.heuristics.AbstractHeuristic method), 20
reset_all() (baal.ModelWrapper method), 13
reset_fcs() (baal.ModelWrapper method), 13
reset_labeled() (baal.active.ActiveLearningDataset method), 17
ResetCallback (class in baal.utils.pytorch_lightning), 22

S
state_dict() (baal.active.ActiveLearningDataset method), 17
state_dict() (baal.ModelWrapper method), 13
step() (baal.active.ActiveLearningLoop method), 17

T
test_on_batch() (baal.ModelWrapper method), 13
test_on_dataset() (baal.ModelWrapper method), 14
train() (baal.ModelWrapper method), 14
train_and_test_on_databases()  
(baal.ModelWrapper method), 14
train_on_batch() (baal.ModelWrapper method), 14
train_on_dataset() (baal.ModelWrapper method), 15