Active Learning: A Systematic Investigation

Full Stack Deep Learning Spring 2021 Capstone Project

Matthias Pfenninger (https://www.linkedin.com/in/matthiaspfenninger/)
Stefan Josef (https://www.linkedin.com/in/stefan-j-7a5a6b120/)
Ravindra Bharathi (https://www.linkedin.com/in/sravindrabharathi/)

April & May 2021

1. Introduction

In recent years, Deep Learning has significantly pushed the boundaries of what is possible with Machine Learning-based systems, especially in the fields of computer vision and natural language processing, and has shown enormous potential for real-world applications. However, in order to achieve such high levels of accuracy, these algorithms require large amounts of annotated training data. Even though the rise of transfer learning has already lowered the amount of annotated data necessary to train high-accuracy models, there is still plenty of room for making the learning process more data-efficient.

In real-world ML projects, data annotation budgets are limited while in many cases plenty of unlabelled examples are available. The difficulty however is, that we don't know a-priori which examples will have the largest impact on model performance and should be labelled within our limited budget. Active Learning tries to solve this issue by putting the model into the annotation process and asking the model what examples should be labelled next.

In our final project for the Full Stack Deep Learning Spring 2021 online course, we focus on investigating various active learning techniques in a practical setting, following the research idea proposed by Scale AI in the course's project guidelines. For our project we decided to mainly focus on the setting of fine-tuning a pre-trained ResNet-50 on the DroughtWatch dataset from the Weights & Biases benchmarks. While we mainly focused on DroughtWatch, we also ran some experiments on MNIST, Cassava Leaf Disease and Deepweeds datasets, to a) include a more basic and well-explored dataset and b) try more datasets .

Since all our team members were entirely new to Active Learning we started the project with a reading phase, where each of us tried to read up on the basics as well as recent developments in the field. After that we entered a coding phase, where each of us explored the dataset and tried to implement different sampling methods from scratch or experiment with existing libraries. In the third project phase we combined our efforts and implemented an Active Learning pipeline as well as a variety of different sampling methods into the course's lab codebase. The main reason for adapting the existing codebase instead of writing our own or just running our experiments in Jupyter notebooks was to maximize our learning from the course's labs by actively modifying and extending the codebase.

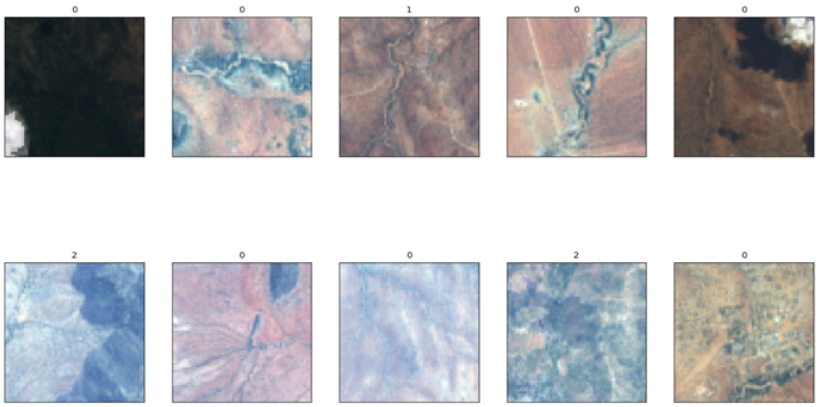
In the final project phase we ran a selection of experiments in different settings and tracked our results in Weights & Biases. The detailed results of our experiments are presented in the results section of this report.

2. Datasets

Following is a quick introduction of the datasets we used in our project. Note that we implemented MNIST, Cassava Leaf Disease and Deepweeds in separate branches.

2.1. DroughtWatch

For testing our Active Learning pipeline and running most experiments, we chose the DoughtWatch dataset from the Weights & Biases benchmarks.



The dataset consists of 86’317 train and 10’778 validation set satellite images from Northern Kenya. We didn’t use the non-public test set of an additional 10’774 images in our experiments. The task is to predict drought conditions based on expert labels that describe how many cows the location at the center of each image can support. The labels are divided into 0, 1, 2 or 3+ cows per image. We decided to explore both the original multi-class classification task as well as a simplified binary classification task grouping the labels into 0 or 1+ cows per image. The satellite images themselves are 65x65 pixels with 11 channels. Here, we also decided to explore both the whole number of channels as inputs to our model as well as reducing the number of channels to the three RGB channels.

The DroughtWatch dataset is certainly a challenging dataset: A) It is not clear a-priori what is the optimal number of channels to be included. B) A very large area (1.95 kilometers across) is compressed into a relatively small resolution image. The area of interest in each image is actually not much larger than a single pixel. C) Unlike many other image classification datasets, this task cannot be solved by human experts by looking at the satellite image alone. Experts were standing in the actual location and were tasked to label the location within 20 meters around them.

For each experiment and Active Learning iteration we use the full validation set and we randomly divide the original training set into initial training set and unlabelled pool. We vary the initial training set sizes, as well as the number of labelled examples that are added to the training set at each Active Learning iterations across experiments.

2.2. MNIST

The MNIST dataset is one of the most popular image classification datasets consisting of 60’000 training and 10’000 test images, that are small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9.

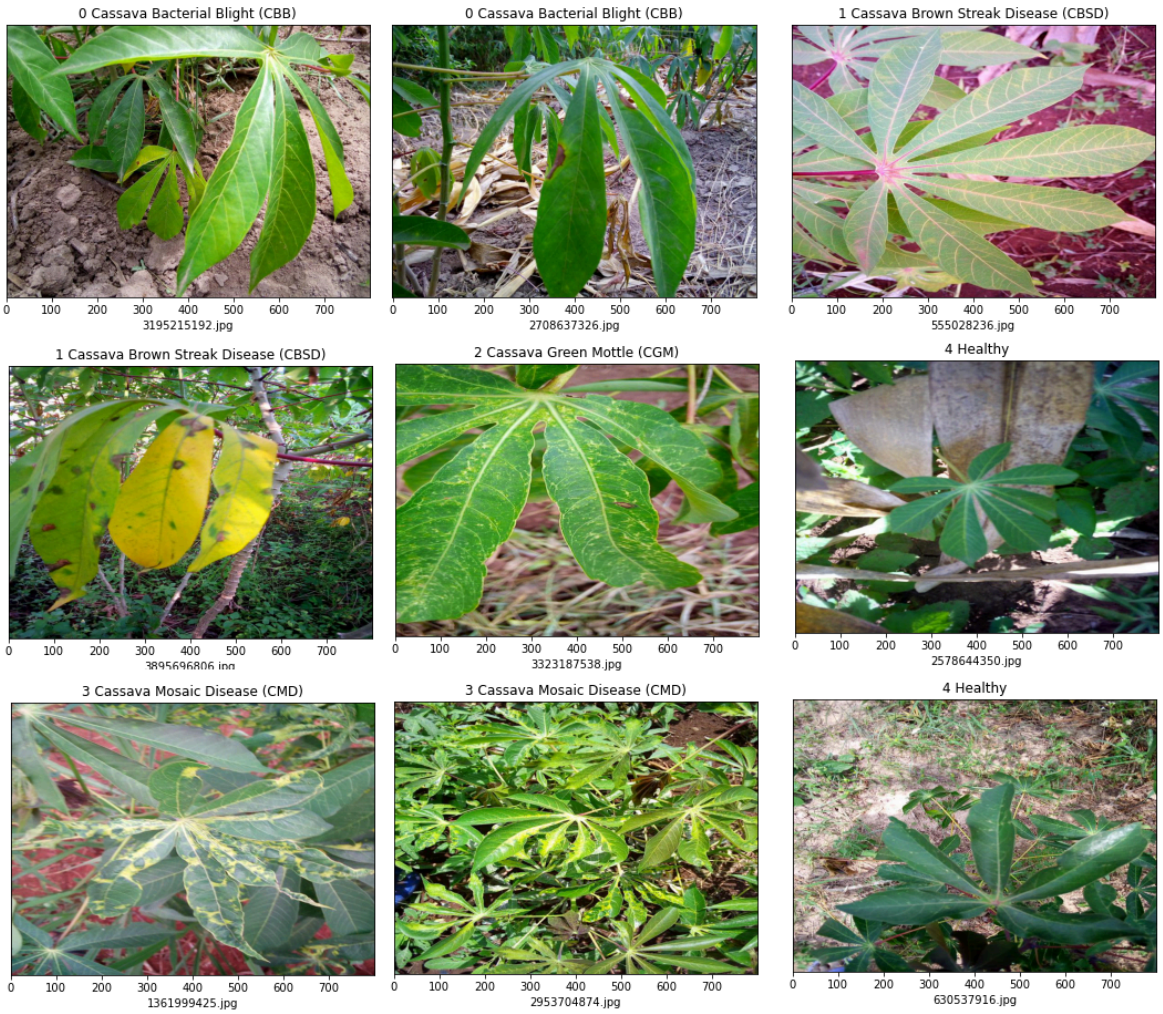
Source: https://en.wikipedia.org/wiki/MNIST_database

We use the dataset in addition to DroughtWatch because it is a much more established dataset with well known, high accuracies that can be achieved.

In our experiments, we only use the training set of 60’000 images and divide it randomly into training, validation and active learning pool datasets. The test set of 10’000 samples is not actively used in the active learning routine.

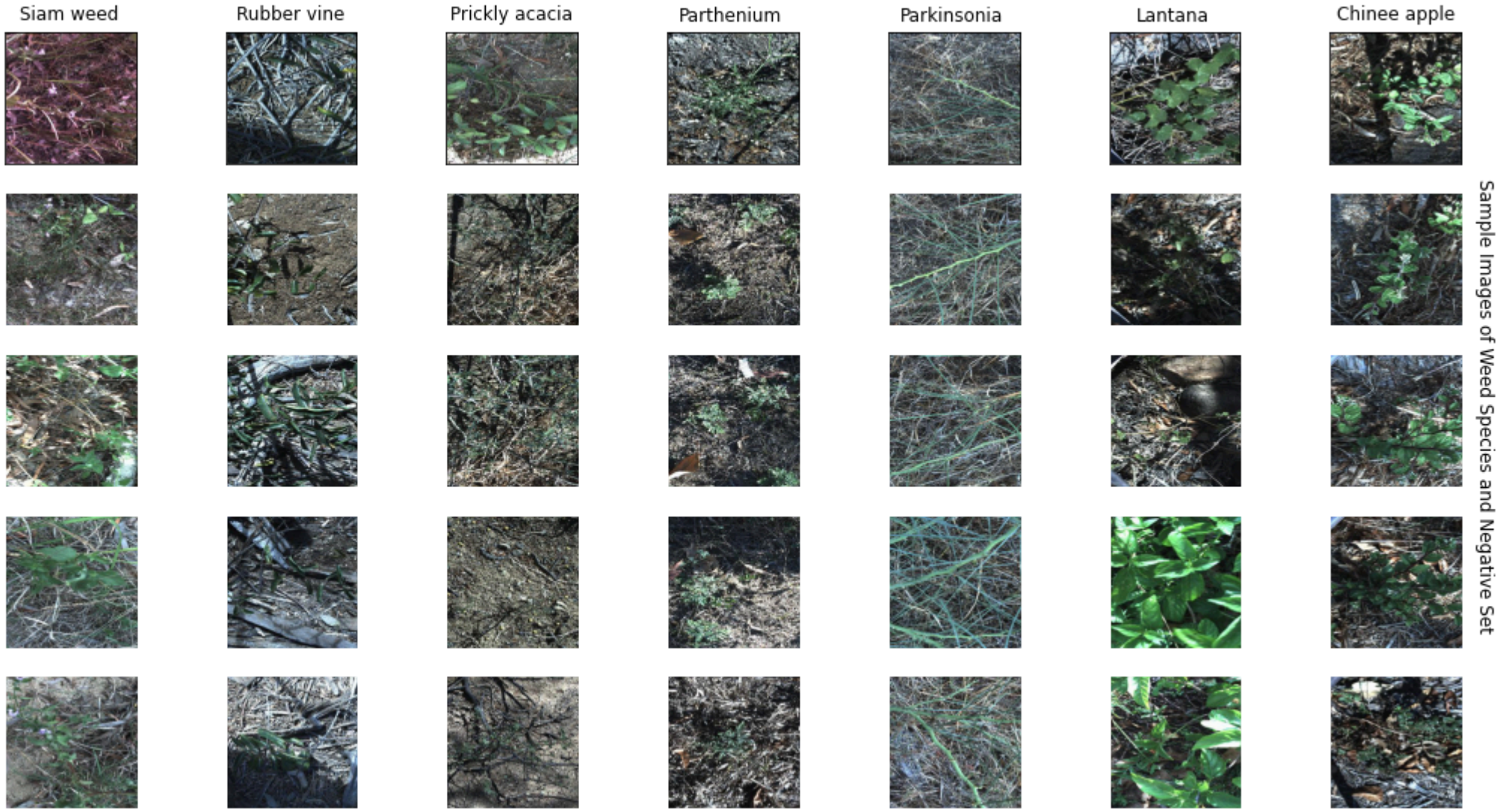
2.3. Cassava Leaf Disease Classification Dataset

The Cassava Leaf Disease Classification Dataset consists of “21,367 labeled images collected during a regular survey in Uganda. Most images were crowdsourced from farmers taking photos of their gardens, and annotated by experts at the National Crops Resources Research Institute (NaCRRI) in collaboration with the AI lab at Makerere University, Kampala. This is in a format that most realistically represents what farmers would need to diagnose in real life.” (kaggle) The images are 512x512 pixels with RGB channels. The prediction task is to classify the images into four disease categories and a fifth category indicating a healthy leaf.



2.4. DeepWeeds Dataset

The DeepWeeds dataset is public dataset based on the work, "DeepWeeds: A Multiclass Weed Species Image Dataset for Deep Learning", published with open access by Scientific Reports: https://www.nature.com/articles/s41598-018-38343-3. “The DeepWeeds dataset consists of 17,509 images capturing eight different weed species native to Australia in situ with neighbouring flora.” (Github) The data are 224x224 RGB images. The classes in the dataset are relatively well balanced, with 1.000-1.100 images in each class and approximately 9.000 negative labels.


Siam weed
Rubber vine
Prickly acacia
Parthenium
Parkinsonia
Lantana
Chinee apple
Sample Images of Weed Species and Negative Set

3. Active Learning Pipeline

In this section we will give a brief overview of the Active Learning Pipeline we implemented based on this course's lab codebase. For implementation details, please refer to Appendix A or to our codebase on Github.

3.1. Data

We adjusted the data module in the codebase to support an Active Learning pipeline. Among others, two major changes were a) to add a module that stores and keeps track of the unlabelled pool and b) to add a function that moves examples from the unlabelled pool to the training set at each Active Learning iteration.

3.2. Model

We used a pre-trained ResNet-50 model for most of our experiments. Only in one of our experiment settings we ran the Active Learning pipeline without relying on pre-training and training the model from scratch at each iteration. Since the focus of this project was not to try the latest state-of-the-art computer vision models, but instead focus on Active Learning, we decided to use ResNet-50 pre-trained on ImageNet because it is a well-established model that has shown robust performance on a variety of image classification tasks.

We adapted the model so that it can handle a flexible amount of input channels, e.g. RGB or 11 channels. Since ResNets usually don't include dropout modules, we added a 3-layer classification head with batchnorm, relu activation functions and dropout in order to prepare the model for experiments with Monte Carlo Dropout. Moreover, we added a model path to extract activations from intermediate layers that some sampling methods needed access to.

3.3. Pipeline

In order to explain our Active Learning pipeline and its flexibility, we describe a step-by-step example of a possible experiment. At the start of each experiment we define the size of our initial training set, the number of Active Learning iterations, the maximum number of epochs within each iteration, if we want to run multi-class or binary classification, all channels or RGB only, as well as other parameters like early stopping, learning rate finder, if a pre-trained model should be used and most importantly, which Active Learning sampling method will be used for sampling new data at each iteration.

The experiment starts by training a model according to the specified parameters on the initial training set, evaluating it on the validation set and logging the best achieved validation set metrics (accuracy of F1 score). Then, at the beginning of each Active Learning iteration we use the model trained in the previous step to sample new labelled examples from the unlabelled pool according to the specified sampling method. Then, we re-initialize the model (we also support the option of continuing training) and train it on the new, expanded training set, evaluate it and log the best metrics. This process continues until we either have reached the maximum number of Active Learning iterations or we exhausted the unlabelled pool.

4. Active Learning Techniques

In this section, we describe all Active Learning sampling techniques that we implemented during the project.

4.1. Baseline: Random

As a baseline, we sample randomly from the unlabelled pool at each iteration.

4.2. Uncertainty Sampling Strategies

Uncertainty sampling is a very common Active Learning technique, where the goal is to sample the examples that the model is most uncertain about. We implemented four different methods according to Robert Monarch's blog. All methods were implemented both in Numpy as well as vectorized versions in PyTorch, which both yield the same results.

- **Least confidence sampling:** This sampling method selects the maximum predicted probability for each example and then samples the k examples with the lowest maximum probability from the unlabelled pool.
- **Margin of confidence sampling:** Selects the two largest predicted probabilities for each example and calculates the difference. Then samples the k examples with the lowest margin from the unlabelled pool.
- **Ratio of confidence sampling:** Similar to margin of confidence sampling, but instead of using the absolute difference, it relies on the ratio between the two highest predicted probabilities. Samples the k examples with the highest ratio, meaning that the second highest probability is very close to the highest probability.
- **Entropy sampling:** This sampling method calculates the difference between all predicted probabilities for an example using the entropy formula. It then samples the k examples with the highest entropy score. Compared to the previous methods, it takes all classes into account, not only the one or two most probable ones.

4.3. Bayesian Uncertainty Sampling Strategies

A shortcoming of the basic uncertainty sampling methods above is that the predicted probabilities that they sample from can in many cases not be trusted. This is especially problematic for Deep Learning models, that often very confidently predict a wrong class. Bayesian neural nets try to overcome this issue by providing better uncertainty estimates. The main idea is to sample a model's weights from a random variable instead having the same deterministic weights for each forward pass. A way to approximate this behavior is Monte Carlo Dropout, where we keep dropout active during inference, run several forward passes and then average the predicted probabilities.

- **Bayesian Active Learning by Disagreement (BALD)**: The idea of this technique is to maximize the information gain by maximising mutual information between predictions and model posterior. The details are depicted in the papers Deep Bayesian Active Learning with Image Data and Bayesian Active Learning for

Classification and Preference Learning.

- **Bayesian least confidence sampling:** Combines Monte Carlo Dropout with least confidence sampling

- **Bayesian margin of confidence sampling:** Combines Monte Carlo Dropout with margin of confidence sampling

- **Bayesian ratio of confidence sampling:** Combines Monte Carlo Dropout with ratio of confidence sampling

- **Bayesian entropy sampling:** Combines Monte Carlo Dropout with entropy sampling

4.4. Diversity Sampling Strategies

Another class of sampling techniques are diversity sampling methods. Instead of sampling the least confident predictions, diversity sampling techniques aim to produce a diverse sample from the distribution in the unlabelled pool. Inspired by the methods in Robert Monarch's blog on diversity sampling, we implemented the four following methods.

- **Model based outliers mean:** This sampling method extracts activations from intermediate layers (in our case from the last three linear layers), then calculates the average activation scores for each layer, and finally also averages across the three layers. Then, we sample the k examples from the unlabelled pool with the lowest average activation scores. The rationale behind this sampling type is that our model is confused by examples with low activation scores due to "lack of information".

- **Model based outliers max:** Same method as above, but instead of taking the mean for each layer and across layers, it takes the maximum activation scores.

- **Combined clustering & outlier based sampling**: Clusters the pool and takes both the most relevant and the most outlier points in each cluster. The idea is to get a set that covers all meaningful trends of the features space as good as possible. Clustering is done with HDBSCAN and outlier scores calculated via GLOSH, both algorithms taken from the hdbscan library.

- **Outlier based sampling**: Takes the samples with the highest outlier scores according to their GLOSH score (calculated via hdbscan library), with the idea to cover points that are not part of any trend of the feature space. We expect that this strategy should be combined with others to perform well, in order to capture both trends and outliers.

4.5. Mixed Sampling Strategies

The goal of mixed sampling strategies is to combine the strengths of uncertainty and diversity sampling and thereby produce a sample where our model is confused while also

reflecting the diversity of the unlabelled pool. In order to produce such mixed sampling strategies it is possible to chain different sampling methods together.

- **Model based outliers mean + least confidence sampling:** This method first samples 4 times the sample size from the unlabelled pool using least confidence sampling. Then the actual sample size is sampled from this pre-selected pool using the model based outliers mean strategy.
- **Model based outliers mean + entropy sampling:** First samples 4 times the sample size from the unlabelled pool using entropy sampling. Then samples the actual sample size from this pre-selected pool using model based outliers.

4.6. Advanced Sampling Strategies

What we call “advanced” sampling techniques are methods that train another model in the sampling step. This model is then used to run inference on the unlabelled pool and sample the examples that will be added to the training set for the next Active Learning iteration.

- **Active Transfer Learning for uncertainty sampling:** This method, presented in Robert Monarch’s blog on active transfer learning, combines the ideas of Active Learning and Transfer Learning. The main idea is to ask the model to predict its own errors. After training the model on our target task with a given training set, we create a new dataset from the validation set where the labels are whether the model predicted an example correctly or not. We then fine-tune the classification head of the previous model on this new dataset and run inference on the unlabelled pool. The examples with the highest probability of being incorrect are sampled.
- **Discriminative Active Learning:** Discriminative Active Learning (DAL) follows a similar process as Active Transfer Learning for uncertainty sampling, but fine-tunes the discriminative model on a different dataset. More specifically, we create a new dataset consisting of examples from our current training set and examples from our unlabelled pool. The model’s classification head is then fine-tuned to predict if an example is part of our training set or unlabelled pool. Finally, we are sampling the examples from the unlabelled pool that have the highest probability according to the fine-tuned model of belonging to the unlabelled pool.

5. Results

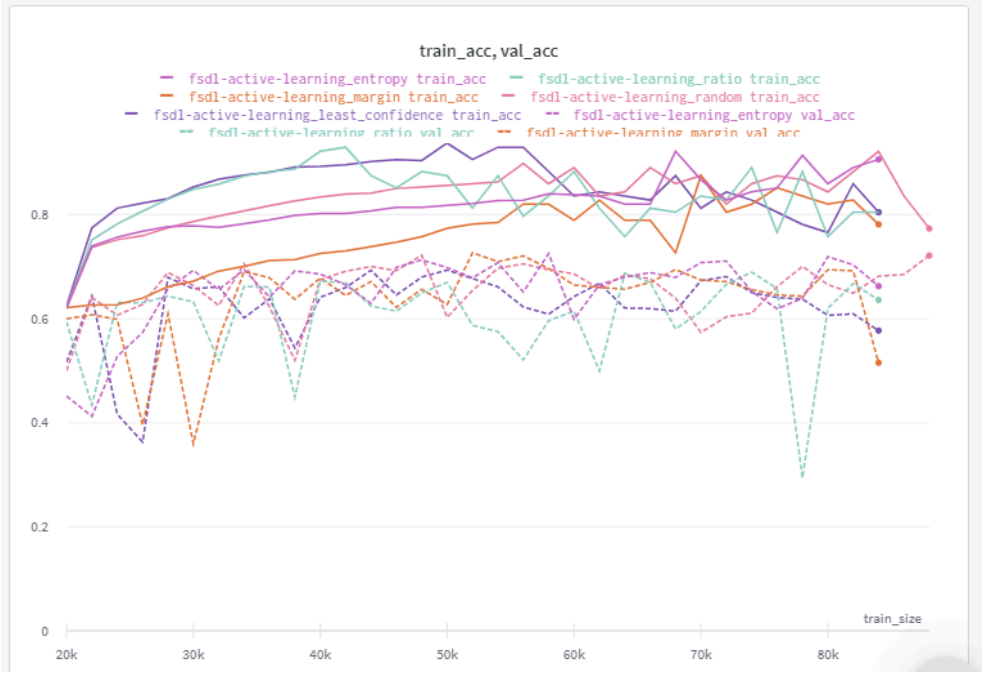
In the following part we present the results of some experiments we ran. Our initial plan was to run all the active learning methods we implemented across all four Droughtwatch scenarios (binary/multi-class, RGB/11 channels) from approximately 25% initial training set size up to the full training set size with a step size of 2.000 additionally labelled examples at each active learning iteration. However, due to time constraints and limitations of our Colab Pro accounts, we didn’t manage to run the number of experiments we set out to do.

Moreover, once we realized that the results in this experimental setting were a) not what we expected to see, i.e. our sampling methods didn’t consistently outperform the random baseline and b) not very insightful, i.e. there was no clear difference among different sampling strategies, we decided to pivot our experimentation strategy and instead run

experiments in many different settings: using different initial training set sizes, different numbers of epochs, different step sizes per active learning iteration, different training strategies (training from scratch, fine-tuning the whole model, fine-tuning only the classification head) and ultimately even different datasets.

5.1. DroughtWatch

Expecting to see a clear difference between random and other strategies, we started to collect training & validation accuracies for the DroughtWatch dataset, starting with an initial training set of 20'000 (approx. 25% of all available samples) and expanding it by 2'000 labelled examples in each iteration, until the whole training set is consumed:



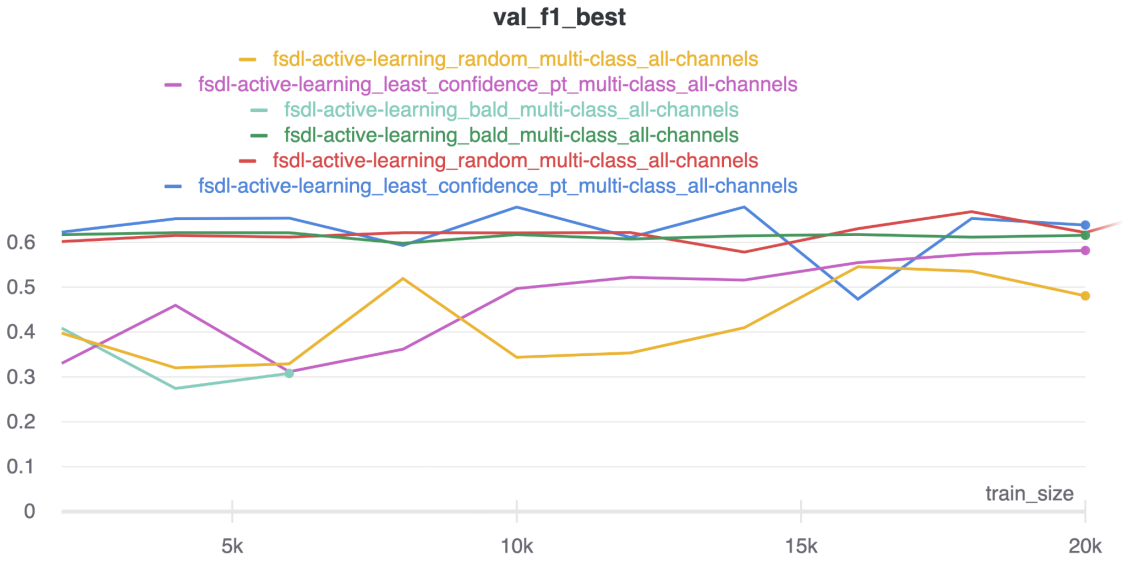
Two surprising observations in these results:

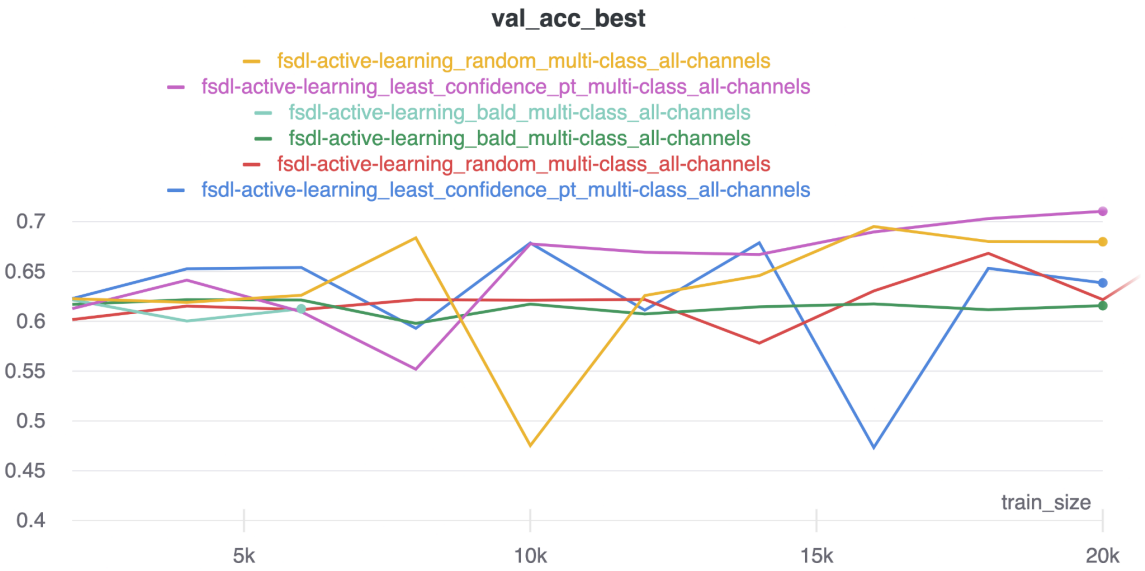
- The validation accuracies start at a very high level with the initial training set (between 0.5 and 0.6) and don't steadily increase, but more or less stay within 0.5 and 0.7. A possible explanation for this behavior could be that using a pre-trained model greatly reduces the need for labelled data and already gives good performance on smaller dataset sizes.
- All the strategies perform similarly and not better than the random baseline.

Since the leaderboard of the W&B DroughtWatch benchmark contains scores of maximum 0.78, we decided to start with smaller initial training set sizes to hopefully see a difference in how fast strategies get to scores around 0.7.

Due to the class imbalance in the data set, we decided to additionally report the F1 score going forward as well, which gives more reasonable results in imbalanced scenarios.

Starting with a training set of 2'000 and increasing by 2'000 up to 20'000 and reporting both F1 and validation accuracy for six different strategies, we see the following:



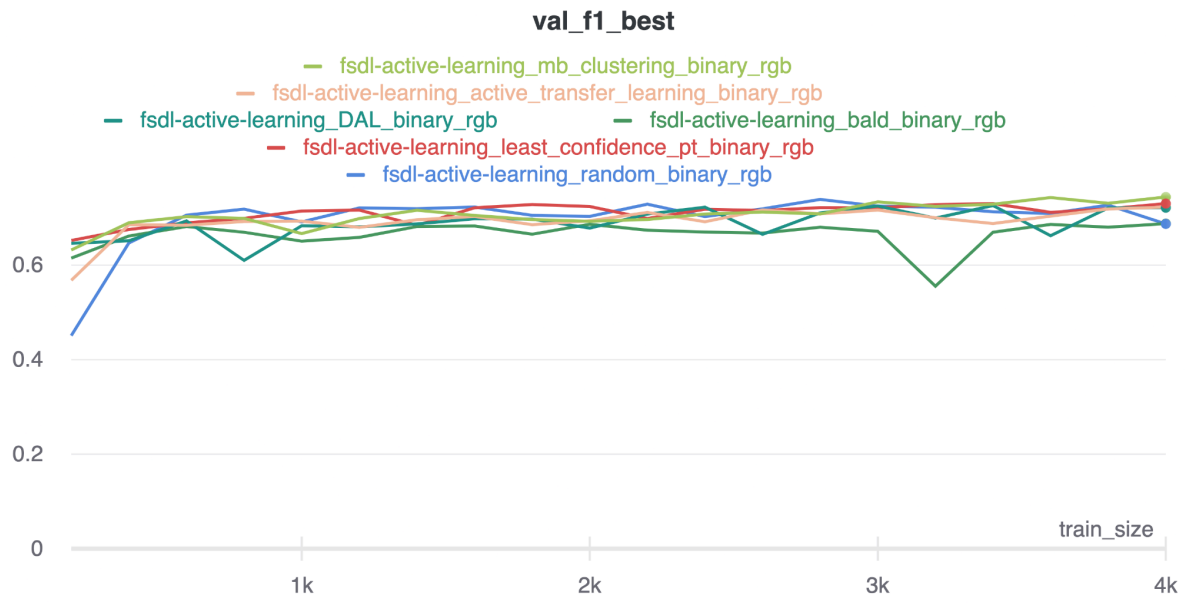


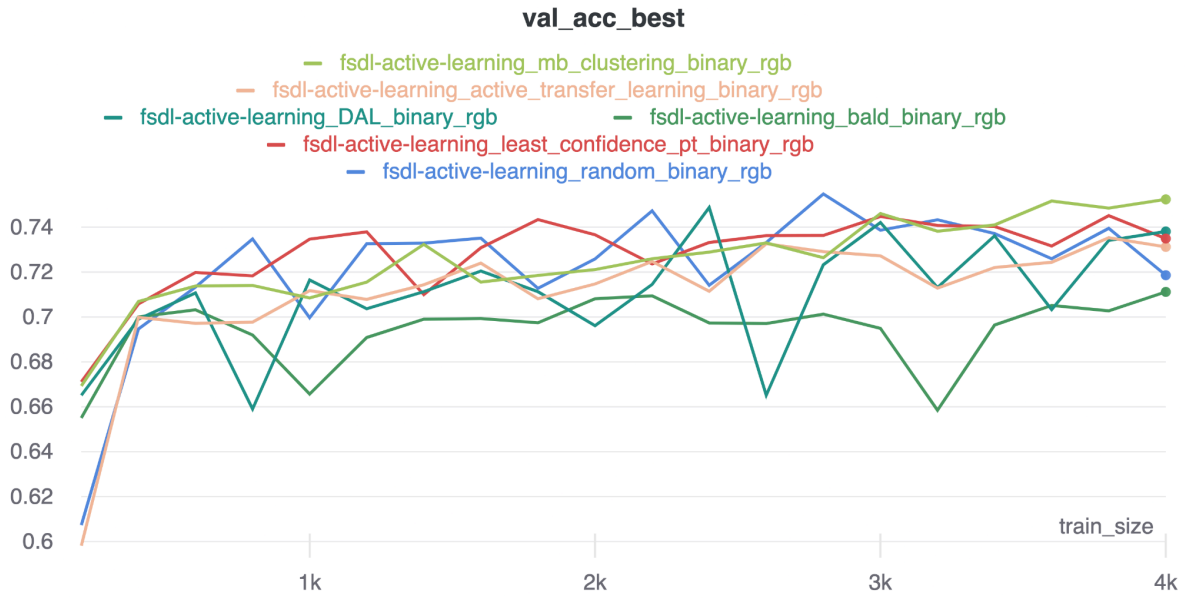
Again in these results and for neither of the two metrics, the same points as before hold:

- Random is not clearly worse than other strategies
- Metrics start on a high level and increase at a very slow rate, sometimes even deteriorating with additional training data

Additionally it shows the effect of random initialization in our experiments. We launched both random and least confidence sampling twice and while they perform very similarly to each other in each run, the two runs themselves differ strongly.

To avoid the challenges of multi-class classification and avoid potential fitting to noise in the 11 image channels, we ran similar experiment scenarios in the simplified setting of binary classification and only using the RGB channels, additionally starting at a smaller training set size (200) and increasing less per iteration (200):

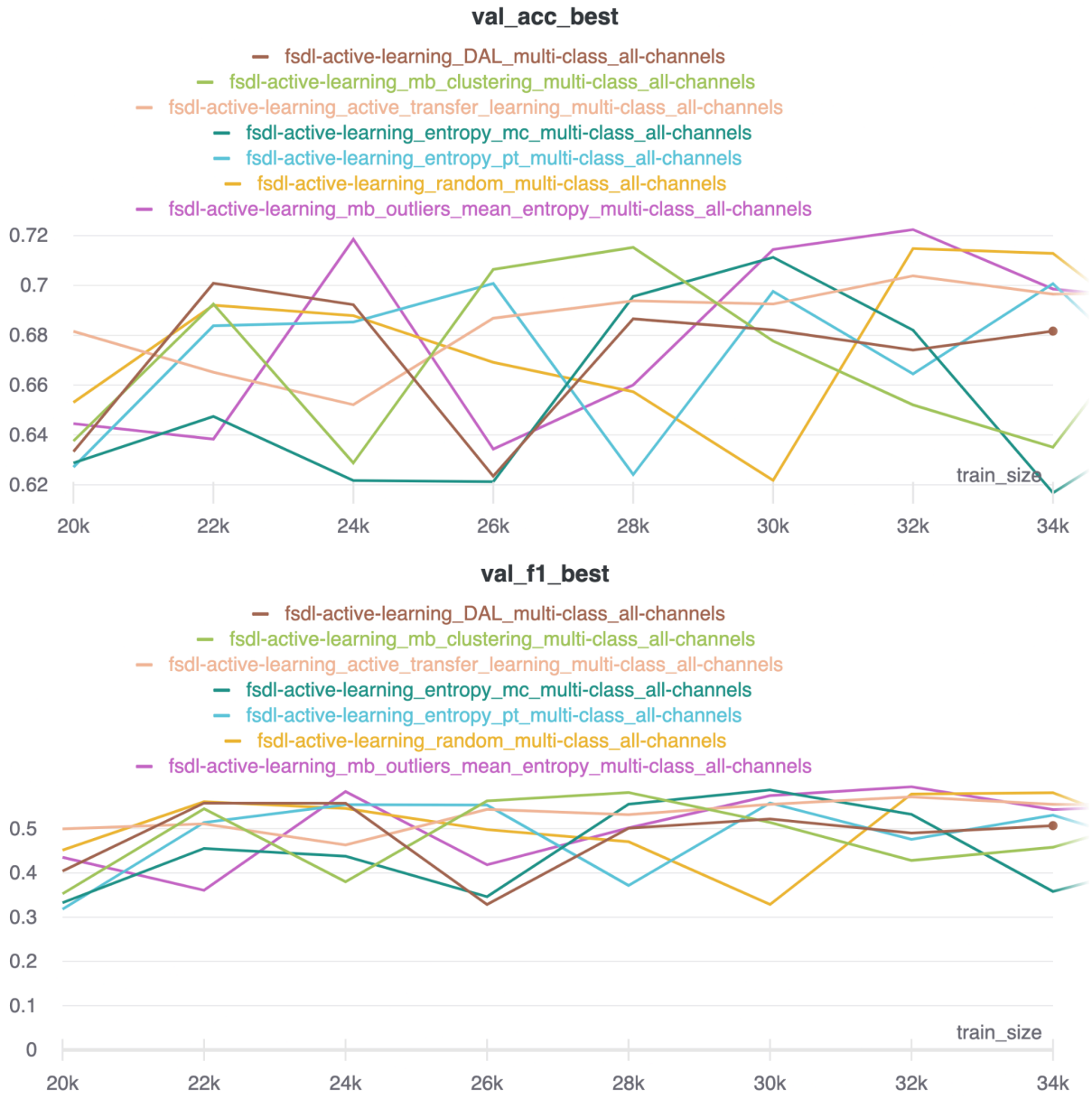




Still the results behave similarly as to the experiments above.

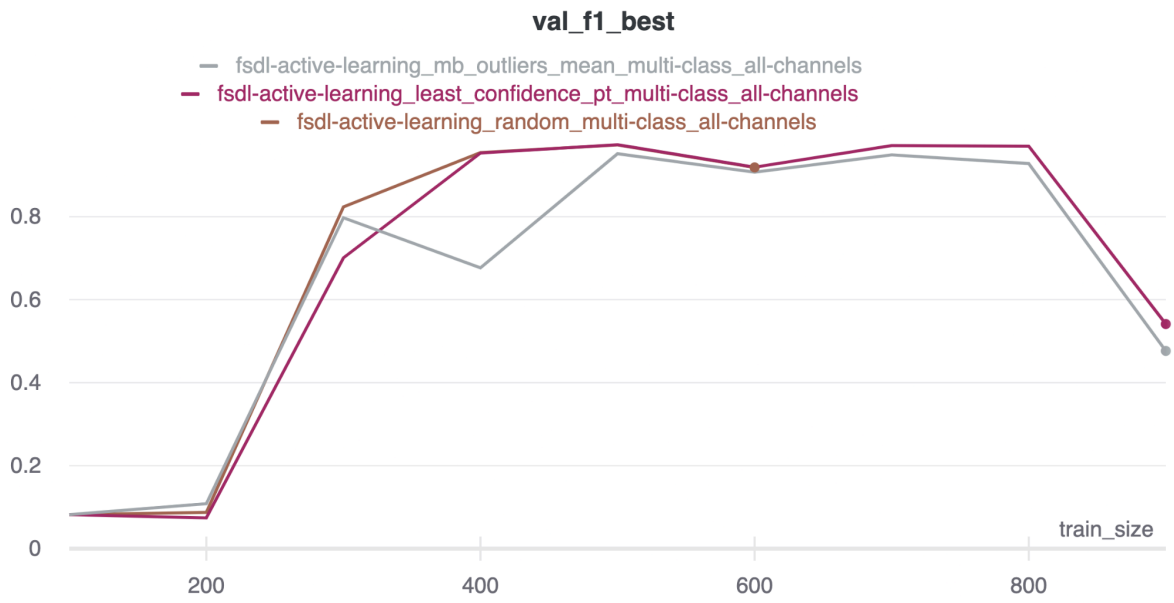
Following our suspicion that transfer learning leads to high levels of accuracy even for small initial training sets, we decided to additionally run experiments where we attempt to train the model from scratch instead of fine-tuning a pre-trained model. Since training from scratch requires large amounts of data, we started these experiments again with an initial training set size of 20'000 and a sample size of 2'000. Another change we made here was to use a fixed learning rate of 3e-4 instead of running learning rate finder at each iteration.

However, even in this experimental setting without pre-training, the results didn't show any clear difference between the random baseline and our sampling methods as well as among the individual sampling methods.



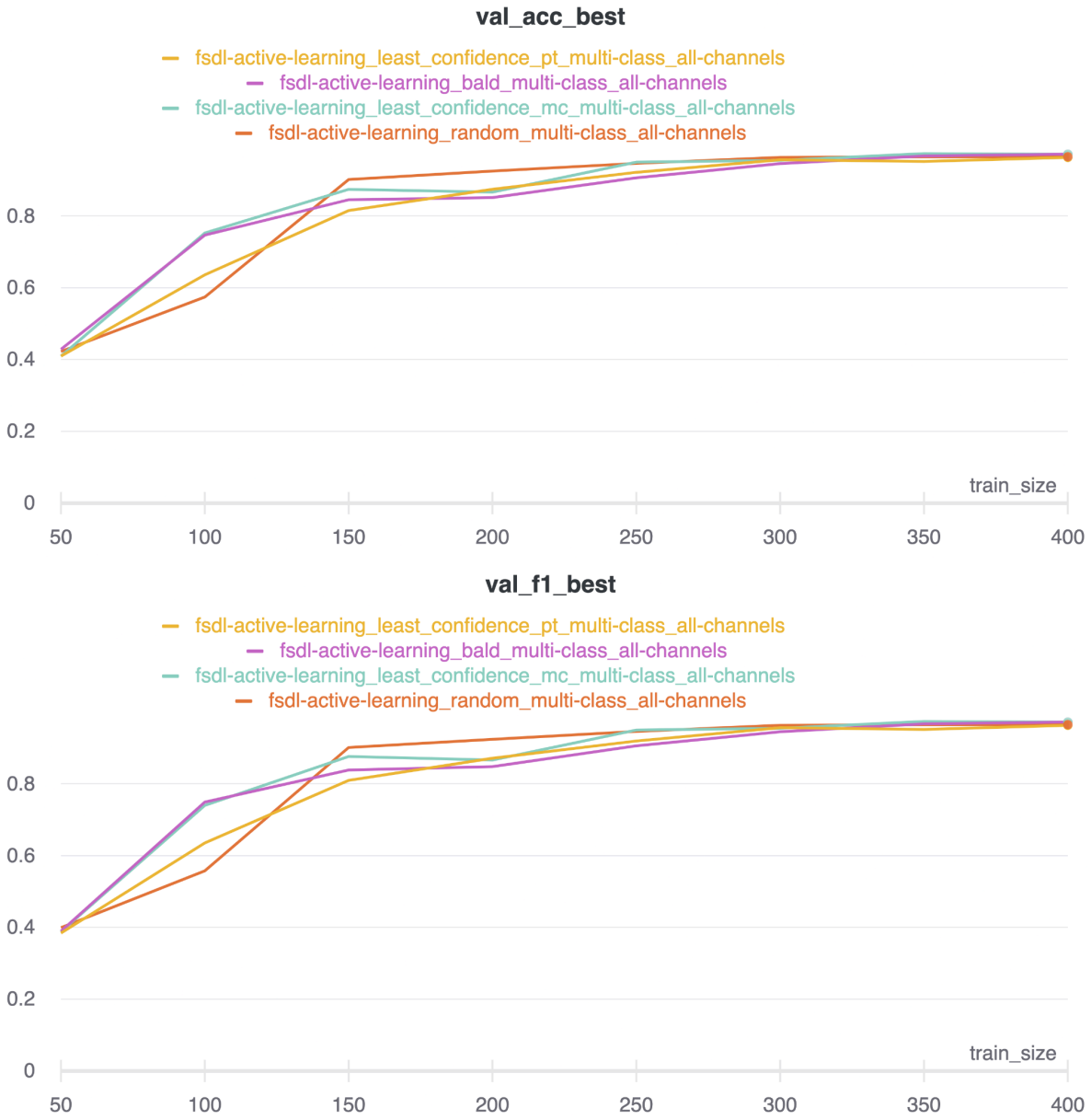
5.2. MNIST

Concerned that the random results described above could be caused by the dataset, we ran similar experiments with MNIST. To avoid having negative impacts from suboptimal learning rates, we additionally used PyTorch Lightning's automatic learning rate finder and started from even smaller training size of 100, increasing by 100 in every iteration:



The results show a somewhat strange training behavior that could be caused by the automatic learning rate that we used in this setting. The fact that the model didn't learn anything in the first iteration (up to training size 200) and drops steeply at training size 900 could be explained by an overly large learning rate. Hence we returned to a fixed learning rate of `3e-4` for the following experiments.

Starting with 50 training samples, increasing by 50 until a total size of 400:



We see the expected pattern of achieving better metrics with a growing training size, but random is not clearly performing worse than the other sampling strategies.

To zoom in on the phase where accuracies are increasing strongly, we started another experiment with initial training set size of 8 and increasing by 4 in each iteration:

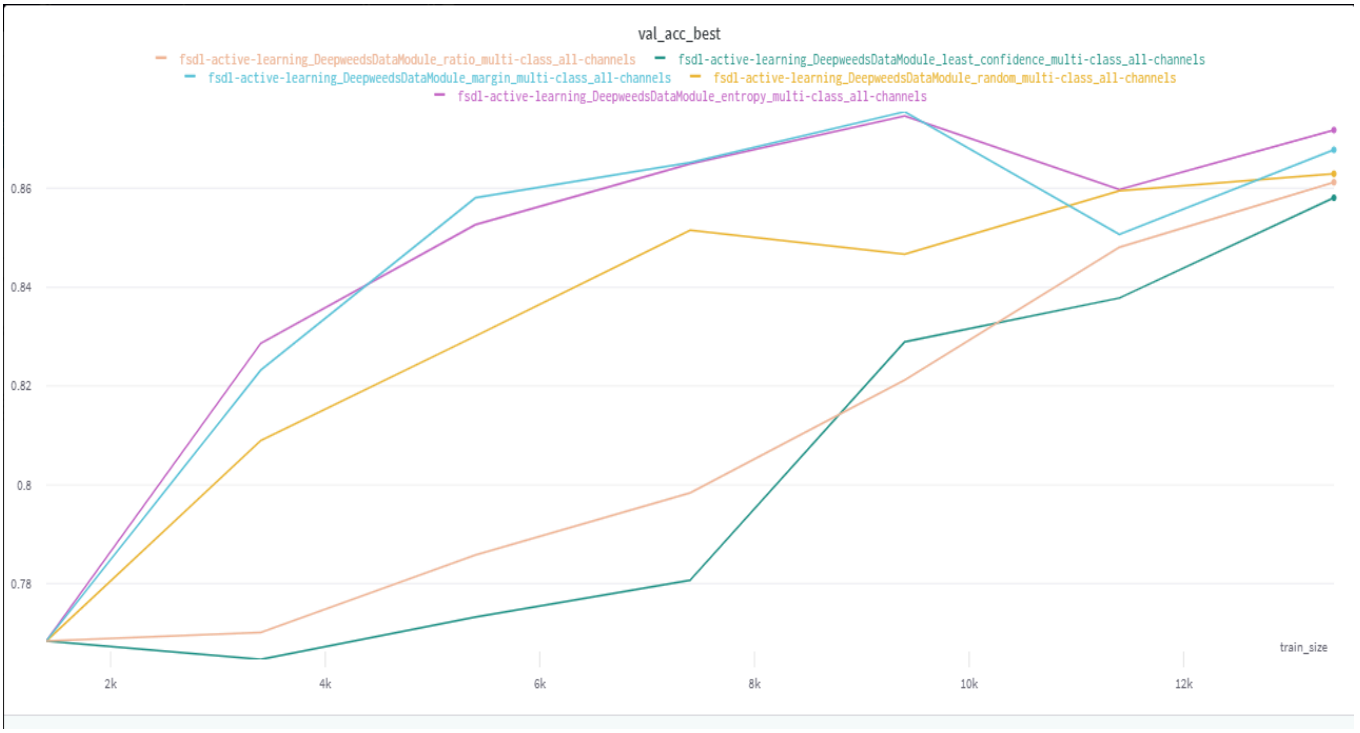
Also here in this small data setting, where we can see the overall increase in accuracy more clearly as we are adding more training examples, there is again no clear winner among the different sampling strategies. While active transfer learning for uncertainty sampling seems to be performing slightly better than the other sampling strategies (incl. random baseline), it is also experiencing two very steep drops in accuracy and it would be a stretch to conclude that it is consistently outperforming the other sampling strategies.

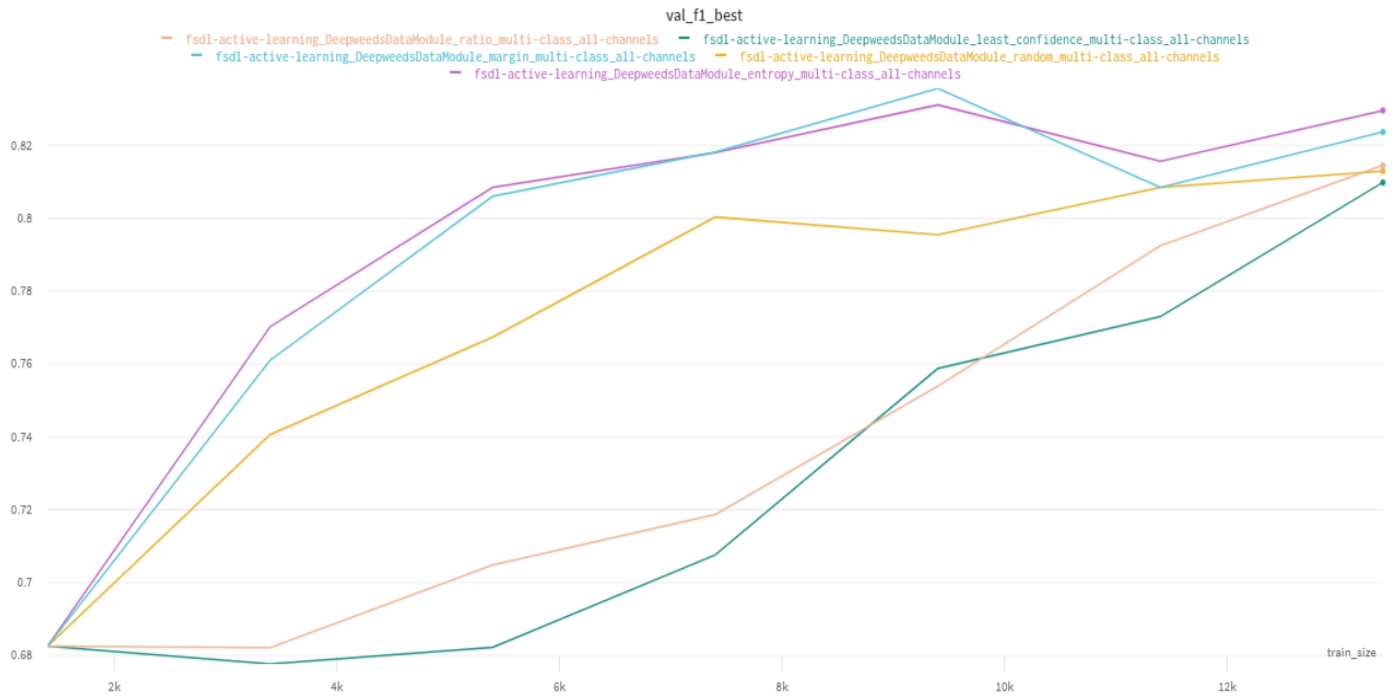
5.3. Deepweeds

For the Deepweeds dataset, we used a ResNet-50 backbone with pretrained weights from ImageNet. We added a couple of fully connected layers as the classification head. The ResNet-50 backbone layers were kept frozen during fine-tuning.

The dataset was split into a 20% validation set and the rest used for active learning unlabelled pool. The initial training set was set to 10% of the unlabelled pool and the training set size was increased by 2000 for every active learning iteration.

Uncertainty sampling methods - least confidence, margin, ratio and entropy, were tested against the baseline of random sampling. For this dataset and this experimental setting, entropy and margin sampling methods clearly showed better gains in validation accuracy and F1 score compared to the baseline random sampling. Surprisingly, least confidence and ratio of confidence sampling performed even worse than the random baseline. It is also interesting to observe that the difference in accuracy kept increasing during the first half of the experiment, but started to get smaller again during the second half.



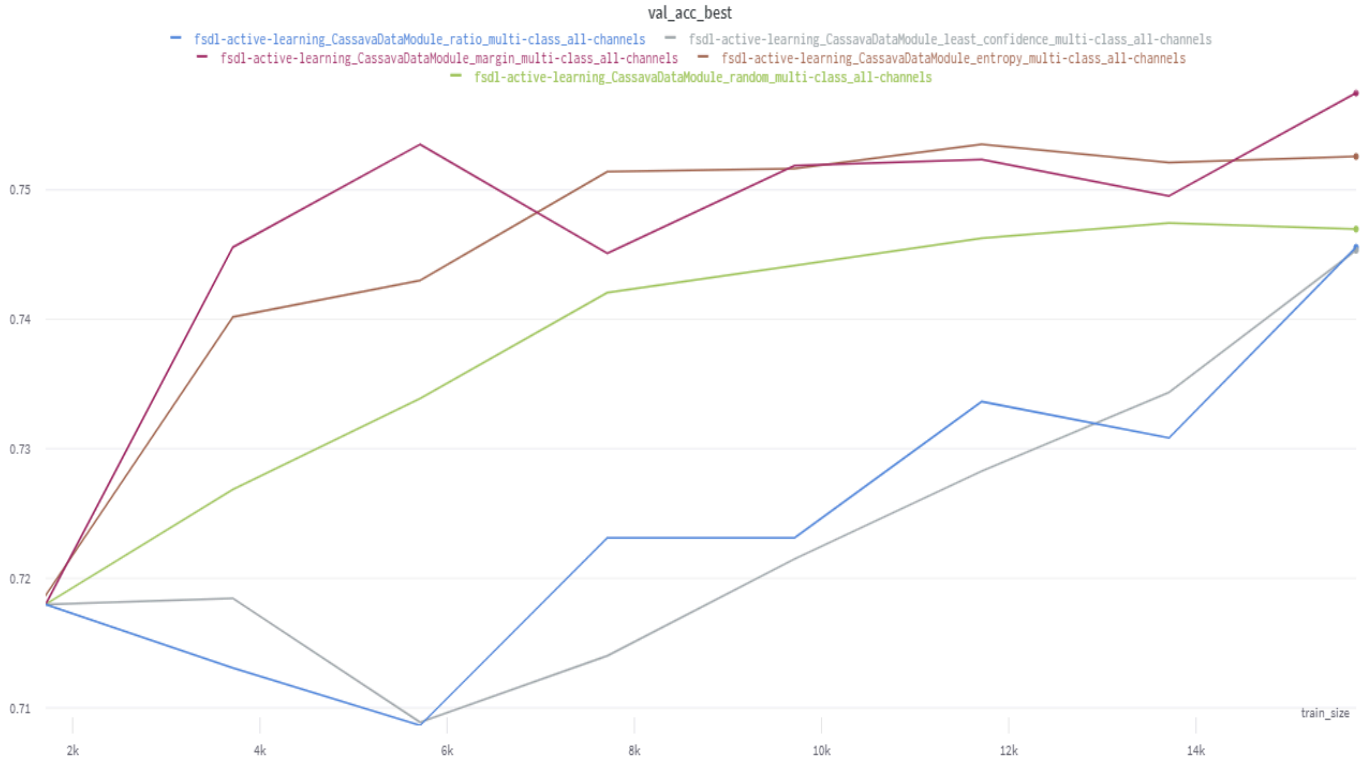


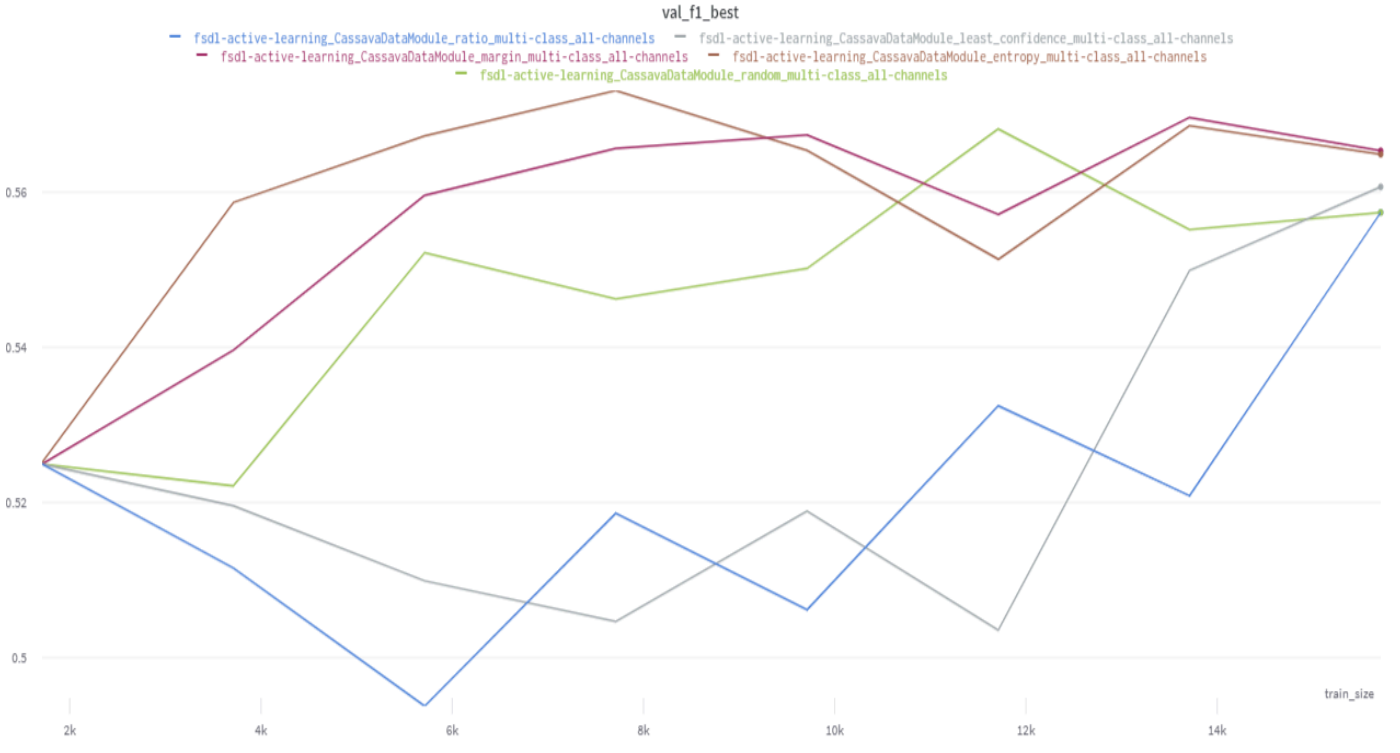
5.3. Cassava Leaf Disease Classification Dataset

For the Cassava Leaf Disease Classification dataset, we used a ResNet-50 backbone with pretrained weights from ImageNet. We added a couple of fully connected layers as the classification head. The ResNet-50 backbone layers were kept frozen during fine-tuning.

The dataset was split into a 20% validation set and the rest used for active learning unlabelled pool. The initial training set was set to 10% of the unlabelled pool and the training set size was increased by 2000 for every active learning iteration.

Uncertainty sampling methods - least confidence, margin, ratio and entropy, were tested against the baseline of random sampling. Similar to the Deepweeds dataset, for this dataset and this experimental setting, entropy and margin sampling methods clearly showed better gains in validation accuracy and F1 score compared to the baseline random sampling. Least confidence and ratio of confidence sampling performed worse than the random baseline. We once again observed that the difference in accuracy kept increasing during the first half of the experiment, but started to get smaller again during the second half as the labelled training set size increases.


val_acc_best
fsdl-active-learning_CassavaDataModule_ratio_multi-class_all-channels
fsdl-active-learning_CassavaDataModule_least_confidence_multi-class_all-channels
fsdl-active-learning_CassavaDataModule_margin_multi-class_all-channels
fsdl-active-learning_CassavaDataModule_entropy_multi-class_all-channels
fsdl-active-learning_CassavaDataModule_random_multi-class_all-channels
0.75
0.74
0.73
0.72
0.71
train_size
2k
4k
6k
8k
10k
12k
14k


val_f1_best
fsdl-active-learning_CassavaDataModule_ratio_multi-class_all-channels
fsdl-active-learning_CassavaDataModule_least_confidence_multi-class_all-channels
fsdl-active-learning_CassavaDataModule_margin_multi-class_all-channels
fsdl-active-learning_CassavaDataModule_entropy_multi-class_all-channels
fsdl-active-learning_CassavaDataModule_random_multi-class_all-channels
0.56
0.54
0.52
0.5
train_size
2k
4k
6k
8k
10k
12k
14k

6. Conclusion & Outlook

To sum up our project, we have implemented an Active Learning pipeline for image classification with 18 different sampling methods (incl. random baseline) covering basic as well as Bayesian uncertainty sampling, diversity sampling, mixed uncertainty and diversity sampling and finally advanced sampling based on transfer learning. We implemented four different datasets to test our pipeline and ran a variety of experiments in different settings.

In our main experiments with DroughtWatch and the few experiments we launched with MNIST, we didn't observe the clear results we expected to see, i.e. Active Learning methods clearly outperforming the random baseline as well as clear differences between sampling strategies. This could be caused by different factors. The DroughtWatch dataset, that we focused our main attention on, is certainly an unusual and challenging dataset and in hindsight it could have been beneficial to start with a more basic dataset before focusing on a more complex one. Fine-tuning ResNet-50 on DroughtWatch turned out to be very unstable and heavily dependent on the choice of the learning rate. In an Active Learning experiment it is also not obvious what should be the optimal choice of the learning rate, since it could be different for different dataset sizes. In addition, the fine-tuning process for a fixed dataset size showed different results across different runs even with the exact same setting, most likely caused by the random factors in our model, i.e. random initialization of the classification head and dropout. The differences across sampling methods in our experiments could therefore also be caused by random factors and can therefore not solely be attributed to the sampling methods themselves. This limitation could be overcome by running the same experiments multiple times and reporting the average results, significantly larger computational resources would be required to run experiments at this scale.

In our experiments with the Cassava Leaf Disease and DeepWeeds datasets however, we observed clear differences across sampling strategies, with margin of confidence and entropy sampling clearly outperforming the random baseline as well as least confidence and ratio of confidence sampling. Unfortunately, due to time constraints we didn't manage to run more experiments with other sampling methods on these datasets until submission.

The project has been an incredible learning experience for all our team members. We learned about a new research field in a relatively short amount of time, learned to use new frameworks and tools, incl. PyTorch Lightning and Weights & Biases, and implemented our pipeline as well as sampling methods from scratch in the course's codebase. Working in the lab codebase gave us the opportunity to really take it apart, understand how the different modules play into each other and expand it to accommodate our needs, meeting our goal to learn as much as possible from the labs. Another challenge we had to overcome was working with limited computational resources. We ran all our experiments from Google Colab Pro accounts and often had to experience interrupted experiments and lost progress.

We have several ideas of how we could continue our collaboration and our work in the field of Active Learning in the future:

- It would be interesting to implement "batch-aware" sampling methods that might lead to a better performance for Deep Learning models compared to a random baseline

- Deploy our Active Learning pipeline and develop a simple UI in which users can upload images and get back suggestions from the model which images to annotate next based on a sampling strategy they choose
- We developed a couple of modAL extensions that could be added to the library
- Integrate a more modern Active Learning framework like baal from Element AI
- Implement Active Learning methods for the fastai library

7. References

Following is a list of relevant resources we used during our research and implementation:

- Uncertainty Sampling Cheat Sheet by Robert Monarch: https://towardsdatascience.com/uncertainty-sampling-cheatsheet-ec57bc067c0b
- Diversity Sampling Cheat Sheet by Robert Monarch: https://towardsdatascience.com/https-towardsdatascience-com-diversity-sampling-cheatsheet-32619693c304
- Active Transfer Learning with PyTorch by Robert Monarch: https://medium.com/pytorch/active-transfer-learning-with-pytorch-71ed889f08c1
- Discriminate Active Learning by Daniel Gissin, Shai Shalev-Shwartz https://arxiv.org/abs/1907.06347
- Deep Bayesian Active Learning with Image Data by Yarin Gal, Riashat Islam, Zoubin Ghahramani: https://arxiv.org/abs/1703.02910
- Bayesian Active Learning for Classification and Preference Learning by Neil Houlsby, Ferenc Huszár, Zoubin Ghahramani, Máté Lengyel: https://arxiv.org/abs/1112.5745
- modAL: A modular active learning framework for Python3: https://modal-python.readthedocs.io/en/latest/
- The hdbscan Clustering Library: https://hdbscan.readthedocs.io/en/latest/index.html
- DeepWeeds Dataset https://github.com/AlexOlsen/DeepWeeds
- Cassava Leaf Disease Classification Dataset https://www.kaggle.com/c/cassava-leaf-disease-classification/data

8. Appendix A: Implementation Details

We structured our code base similarly to the one that was developed as part of the text recognizer labs in the course. Following is a description of the most important extensions.

8.1. Code Repository

https://github.com/ravindrabharathi/fsdl-active-learning2

8.2. BaseDataModule

The central class that implements on top of PyTorch's `LightningDataModule` is extended with the following main methods:

- `expand_training_set(sample_idx)`: Expand the module's train data with data from the unlabelled pool identified by the `sample_idx` argument.
- `get_activation_scores(model)`: Feed all samples from the unlabelled pool into the `model` provided as argument and get out activations from different intermediate layers inside the model. This method is needed for model based active learning techniques.
- `get_pool_probabilities(model, T)`: Feed all samples from the unlabelled pool into the `model` provided as argument multiple times (denoted by the parameter `T`), while keeping dropout layers in the model activated. This method is needed for Monte Carlo / Bayesian active learning techniques.

Additional helper methods:

- `unlabelled_dataloader`: DataLoader similar to the ones for train/validation/test that can be used to load data from the unlabelled sample pool.
- `get_ds_length(ds_name)`: Helper method to get the size of the different data sets (train/val/test/pool) in the module.
- `enable_dropout(model)`: Set all dropout layers in `model` into train modus, meaning that they are activated and are also applied during prediction.

8.3. Datasets

8.3.1. DroughtWatch

Additionally to the parent BaseDataModule functionality, the following logic is implemented in the `DroughtWatch` data class:

- **Data loading / preprocessing**
  Data is loaded as zipped TensorFlow's TFRecords format from a publicly available URL configured under `data/raw/droughtwatch/metadata.toml`. The loading procedure is divided into multiple steps such that repeated runs only need to rerun

the steps needed (e.g. if you change the split sizes, only steps 2 & 3 need to be executed again).

1. Downloading the data as ZIP to local
2. Unzipping, conversion to regular numpy arrays and storing in train/validation/pool splits in compressed `HDF5` format.
3. Loading the splitted `HDF5` data into the DroughtWatch class.

- **Configurable args parameters**
  The data class is configurable via args parameters `bands` (which of the 11 image bands to use), `rgb` (whether to use only RGB channels) and `binary` (whether to load the dataset as multiclass or binary classification set)

8.3.2. MNIST

Additionally to the parent BaseDataModule functionality, the following logic is implemented in the `MNIST` data class:

- **Data loading / preprocessing**
  Data is loaded as PyTorch Vision dataset, converted to NumPy arrays, split to train/val/pool datasets and assigned to the respective BaseDataModule class attributes. Since the conversion and splitting is rather quick, everything is done in-memory and not further persisted to disk – with one exception: The PyTorch Vision API is designed in such a way so that the raw ZIPs are only downloaded if not available locally already.

8.3.3 Deepweeds

The DeepweedsDataModule extends Pytorch Lightning LightningDataModule and uses a custom Pytorch Dataset (DeepweedsDataset) . It exposes the same methods as the DroughtWatch DataModule. The Images and the labels (csv file) for this dataset are stored in Google drive due to the size of data and copied to the local data folder while running experiments. The images are resized to 224x224 and ImageNet Normalization applied before feeding to a Custom ResNet50 model.

8.3.4 Cassava

The CassavaDataModule extends Pytorch Lightning LightningDataModule and uses a custom Pytorch Dataset (CassavaDataset) . It exposes the same methods as the DroughtWatch DataModule. The Images and the labels (csv file) for this dataset are stored in Google drive due to the size of data and copied to the local data folder while running experiments. The images are resized to 224x224 and ImageNet Normalization applied before feeding to a Custom ResNet50 model.

8.4. Models

A custom ResnetClassifier PyTorch model class was implemented that contains the following main features:

- Core: PyTorch's `ResNet50` TorchVision model
- Input layer: `in_channels` of the first convolutional layer is dynamically adapted based on the `n_channels` args parameter
- Output layer: `out_channels` of the last fully connected layer is dynamically adapted based on the `n_classes` args parameter

Other mentionable adaptations are the following:

- Dropout: If the `dropout` args parameter is activated, the last fully connected layer is replaced by a sequence of linear / ReLU / Dropout / BatchNorm layers. The goal of this is to introduce non-deterministic dropout behaviour in order to use Bayesian/Monte Carlo active learning techniques.
- Pretraining: If the `pretrained` args parameter is activated, the ResNet50 model is used in its pretrained version (provided by the PyTorch Vision API).
- Intermediate activations: The `forward` method is implemented in such a way that it can either return the final activations only, or additionally activations from intermediate layers. This is required in model based active learning techniques that are built on intermediate model outputs.
- RGB/Binary switch: If the args parameters `rgb` or `binary` are activated, input and output layers are automatically adapted accordingly to the required number of channels.

8.5. Metrics

To adequately measure our experiments, we implemented two metrics helper classes:

- `MaxAccuracyLogger(pl.callbacks.Callback)`: A callback compatible with PyTorch Lightning's Trainer interface to keep track of the maximum of a certain metric over multiple epochs: e.g. the best F1 score over 20 epochs of training.

  This callback is needed to plot the *best* metric against a certain other metric in W&B plots, e.g.: Plotting the best F1 score against the size of the training_set at that point.

- `F1_Score(pl.metrics.F1)`: Fixed PyTorch F1 metric that can handle non-normalized prediction inputs.

8.6. Experiment Routine

The experiment routine takes the following most important args parameters:

- `sampling_method`: Active learning sampling technique to use
- `max_epochs`: Maximum of epochs to train per active learning iteration

- `data_class`: Data class to use
- `model_class`: Model class to use
- `n_train_images`: Number of training samples with which model is trained in initial iteration
- `al_samples_per_iter`: Number of training samples that are added to the training set in each iteration
- `al_iter`: Number of active learning iterations

Based on the args Parameter `sampling_method`, the experiment routine automatically decides how to handle each active learning iteration:

- Basic sampling: Pool predictions are calculated with PyTorch's `Trainer` built in testing routine and then passed to the sampling method which returns the indices to add to the training set in the next iteration.

- Monte Carlo/Bayesian sampling: Multiple predictions for every data point in the pool are done via the separately implemented `get_pool_probabilities` that runs the model's `forward` method multiple times. The predictions are then passed to the sampling method which again returns the indices for the next iteration.

- Model based sampling: Activation scores at different intermediate layers of the model are calculated via the externalized `get_activation_scores` method. These are then passed to the respective sampling method which again returns the indices for the next iteration.

- Active transfer learning: For speed up, a subsample of both train data and pool data is taken. We then train a model to distinguish between correct and incorrect predictions based on this reduced train set, and apply it on the reduced pool data to get the points which the highest probability of being predicted incorrectly.

8.7. modAL Integration

A separate experiment running routine was developed that integrates with the modAL active learning library.

It provides similar parametrization possibilities, but conducts the active learning iterations leveraging the modAL framework. This enables to apply modAL's built in classification uncertainty techniques (see documentation for details):

- `uncertainty_sampling`
- `margin_sampling`
- `entropy_sampling`

Additionally, we implemented multiple sampling strategies that can be used as part of the modAL framework in general:

- `random`: Baseline technique to randomly sample from the pool

- `bald`: Active learning sampling technique that maximizes the information gain via maximising mutual information between predictions and model posterior (Bayesian Active Learning by Disagreement - BALD) – for details see chapter about techniques above.
- `max_entropy`: Bayesian entropy sampling that maximizes the predictive entropy – for details see chapter about techniques above.
- `cluster_outlier_combined`: Diversity sampling technique that clusters the pool and takes both the most relevant and the most outlier points in each cluster to get a set that is as diverse as possible. Clustering is done with the HDBSCAN algorithm and outlier scores calculated via GLOSH score, both part of the hdbscan library.
- `outlier`: Diversity sampling technique that takes the samples with the highest outlier scores according to their GLOSH score, calculated via the hdbscan library.

All of them work only by taking as input modAL's `ActiveLearner`, a NumPy array containing `X` values of the pool, and the number of instances to be sampled.

8.8. Examples: How to Run Experiments

8.8.1. DroughtWatch

Run an experiment with the DroughtWatch dataset, starting at 1000 training samples and increasing by 500 for 20 iterations, training for max. 20 epochs in each iteration. Activate the RGB and Binary scenario additionally, set the learning rate and start with a pretrained ResNet model:

```
python training/run_experiment.py \
  --sampling_method=active_transfer_learning \
  --data_class=DroughtWatch \
  --model_class=ResnetClassifier \
  --n_train_images=1000 \
  --al_samples_per_iter=500 \
  --al_iter=20 \
  --max_epochs=20 \
  --pretrained=True \
  --binary \
  --rgb \
  --lr=3e-4 \
  --gpus=1 \
  --wandb
```

8.8.2. MNIST

Run an experiment with the MNIST dataset with all default parameters:

```
python training/run_experiment.py \
```

```
  --data_class=MNIST \
  --model_class=MNISTResnetClassifier \
  --gpus=1 \
  --wandb
```

8.8.3. modAL

Run an experiment using the modAL framework using a margin sampling strategy with the DroughtWatch dataset:

```
python training/run_modal_experiment.py \
  --data_class=DroughtWatch \
  --model_class=ResnetClassifier \
  --al_query_strategy=margin_sampling
  --gpus=1 --wandb
```

8.8.4 Deepweeds

Run an experiment with the Deepweeds dataset . Options for sampling_method are ‘random’,’least_confidence’,’margin’,’ratio’,’entropy’

```
python training/run_experiment.py --gpus=1 --max_epochs=10 --num_workers=4
--data_class=DeepweedsDataModule            --model_class=ResnetClassifier3
--sampling_method="entropy" --batch_size=128
```

8.8.4 Cassava Leaf Disease Classification

Run an experiment with the Cassava leaf disease classification dataset . Options for sampling_method are ‘random’,’least_confidence’,’margin’,’ratio’,’entropy’

```
python training/run_experiment.py --gpus=1 --max_epochs=10 --num_workers=4
--data_class=DeepweedsDataModule            --model_class=ResnetClassifier2
--sampling_method="entropy" --batch_size=128
```