1. TASK DESCRIPTION

In this task, you will implement a *Bayesian Neural Network* that yields well-calibrated predictions based on what you learned in class.

TASK BACKGROUND

BAYESIAN NEURAL NETWORKS

Neural networks have enjoyed significant successes in recent years across a variety of domains. Many practitioners claim that, to bring deep learning into more high-stakes domains such as healthcare and self-driving vehicles, neural networks must be able to report uncertainty in their predictions. An approach for doing this involves using *Bayesian Neural Networks (BNNs)*. BNNs combine Bayesian principles of model averaging with the black-box approach of deep learning. As a result, BNNs' predictions often have better calibrated uncertainty in comparison to traditional neural networks.

During the lectures, you have already seen that one way to learn a BNN (Bayesian Neural Network) via Maximum a Posteriori (MAP) estimation:

$$\hat{\theta} = \arg\min_{\theta} -\log p(\theta) - \sum_{i=1}^{n} \log p(y_i|x_i, \theta).$$

Here, x_i is the training input and y_i is the training label. However, since the posterior and predictive distribution are intractable, we need approximate inference techniques in practice. In this task, you will implement approximate inference via *SWA-Gaussian (SWAG)* (Maddox et al., 2019) (https://arxiv.org/pdf/1902.02476.pdf), an extension of Stochastic Weight Averaging (SWA) (Izmailov et al., 2018) (https://arxiv.org/pdf/1803.05407.pdf). SWAG is a simple method that stores weight statistics during training, and uses those to fit an approximate Gaussian posterior.

CALIBRATION

In what follows, we utilize the notation and terminology of Guo et al. (2015) (http://proceedings.mlr.press/v70/guo17a/guo17a.pdf).

We focus on a supervised learning problem with features $X \in \mathcal{X}$ and labels $Y \in \mathcal{Y}$, where both X and Y are random variables. In this task, $\mathcal{X} = \mathbb{R}^3 \times \mathbb{R}^{60} \times \mathbb{R}^{60}$ are satellite images and $\mathcal{Y} = \{0, \dots, 5\}$ is a discrete label space. Given some $X \in \mathcal{X}$, a neural network $h(X) = (\hat{Y}, \hat{P})$ outputs a tuple consisting of a label $\hat{Y} \in \mathcal{Y}$ and confidence $\hat{P} \in [0, 1]$.

We say that a model is *perfectly calibrated* if its confidence matches its performance, that is,

$$\mathbb{P}(\hat{Y} = Y \mid \hat{P} = p) = p, \quad \forall p \in [0, 1],$$

where the randomness is jointly over X and Y.

While all samples that your network will observe during training are well-defined, some test samples are ambiguous, that is, cannot be assigned to one particular label. Therefore, having a well-calibrated network is important: if such a network does not confidently assign any particular class to a sample, that sample is likely ambiguous.

To further highlight the importance of calibration in practice, consider the task of supporting doctors in medical diagnosis. Suppose that real doctors are correct in 98% of all cases while your model's diagnostic accuracy is 95%. In itself, your model is not a useful tool as the doctors do not want to incur an additional 3 percentage points of incorrect diagnoses.

Now imagine that your model is well-calibrated. That is, it can accurately estimate the probability of its predictions being correct. For 50% of the patients, your model estimates this probability at 99% and is indeed correct for 99% of those diagnoses. The doctors will happily entrust those patients to your algorithm. The other 50% of patients are harder to diagnose, so your model estimates its accuracy on the harder cases to be only 91%, which also coincides with the model's actual performance. The doctors will diagnose those harder cases themselves. While your model is overall less accurate than the doctors, thanks to its calibrated uncertainty estimates, we managed to increase the overall diagnostic accuracy (to 98.5%) and make the doctors' jobs about 50% easier.

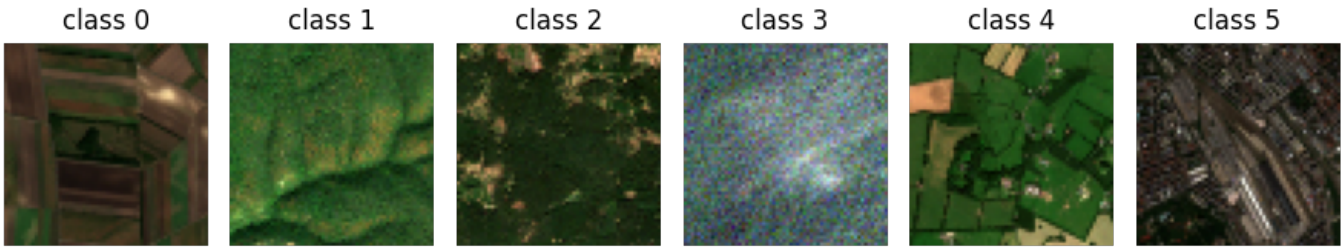
PROBLEM SETUP

DATASET

All features are 60x60 RGB satellite images from various locations. Each image is classified as one or more types of land usage. A challenge when working with satellite images is the huge amount of data, combined with the complexity of processing natural images. Deep learning methods can handle such data. However, working with world-scale data introduces many edge-case and ambiguities, making uncertainty awareness a necessity.

Our training set consists 1800 satellite images, each corresponding to exactly one out of six types of land usage (see below for examples). However, the test set contains a certain fraction of images that correspond to multiple types of land usage, e.g., a body of water surrounded by a forest. Those mixed land usage types may include ones not present in the training set. Furthermore, some images might contain seasonal snow or clouds. The training and validation sets both include information about snowy and cloudy images (see `dataset_train` and `dataset_val` in the `main()` method for details), but the test set does not.

The following are examples of images with a well-defined land usage type:



The following are examples of ambiguous images:

We additionally provide a small validation set, consisting of 20 well-defined images per class, and 20 ambiguous images. You can use the validation samples to further calibrate your model, but not for performing SWAG itself. Also keep in mind that the validation set is very small. Hence, be careful to avoid overfitting, especially to the validation cost and ECE.

YOUR TASK

Your task is to implement SWA-Gaussian to classify land-use patterns from satellite images, and detect ambiguous/hard images using your model's predicted confidence. For each test sample, your method should either output a class, or "don't know". We assign each prediction a cost, depending on the output; see further below for details.

All sub-tasks should be implemented by filling-in the solution template `solution.py`, which can be found in the handout. Your solution should not change any of the other files in the handout. Most of your solution will focus on the `SWAGInference` class, and the few exceptions are marked with `# TODO`. The solution template and runner already handle auxiliary tasks such as data loading so that you can focus on implementing SWAG and calibration.

The concrete sub-tasks are as follows:

1. Familiarize yourself with the SWAG method in Maddox et al., 2019 (https://arxiv.org/pdf/1902.02476.pdf), and skim the

original SWA paper (Izmailov et al., 2018) (https://arxiv.org/pdf/1803.05407.pdf) for some background and tricks.
2. Implement SWAG-Diagonal. You can search the solution template for comments beginning with `TODO(1)`; those guide you through the implementation. You can ignore all other `TODO` comments for now. A correct implementation with the hyperparameters in the solution templates should pass the easy baseline.
3. Implement full SWAG and improve your model's calibration/predictions. Besides implementing full SWAG, you can choose how you want to improve your cost. Possible approaches are post-hoc calibration methods, improving your mapping from predicted confidences to labels, and tweaking SWAG hyperparameters. Comments beginning with `TODO(2)` in the solution template provide pointers for possible extensions.

Important: Working with neural networks can require a lot of computational power. We designed this task such that you can beat the hard baseline using an algorithm that runs for a small number of minutes on the CPU. In particular, beating the hard baseline requires *at most* as many SWAG epochs and Bayesian model averaging samples as the numbers provided in the solution template, and should be done using the small pretrained network we provide (that is, no changes to model architecture or additional MAP inference).

Note that the template in `solution.py` uses *PyTorch* as its neural network library. If you are new to PyTorch, you might want to check out the PyTorch quickstart guide (https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html).

METRICS

Your task is scored based on a cost function applied to your predictions (called *prediction cost*). Additionally, to highlight the importance of calibration, we measure your model's calibration using the *expected calibration error (ECE)*, and add a penalty to the cost if the error is too large.

Remember that for predictions $\hat{Y}$ with confidences $\hat{P}$, your model is well-calibrated if $\mathbb{P}(\hat{Y} = Y \mid \hat{P} = p) \approx p$, $\forall p \in [0, 1]$. An obvious error measure is the expected absolute difference between the true accuracy $\mathbb{P}(\hat{Y} = Y \mid \hat{P} = p)$ and the confidence p, that is,

$$\mathbb{E}_{p \sim \hat{P}} \left[\left| \mathbb{P}(\hat{Y} = Y \mid \hat{P} = p) - p \right| \right].$$

This criterion is called the *Expected Calibration Error (ECE)*.

However, we do not know the true accuracy $\mathbb{P}(\hat{Y} = Y \mid \hat{P} = p)$ in practice. In this task, we approximate the true accuracy via quantization over M intervals, leading to the *empirical accuracy*. For $m \in [M]$, let B_m denote the set of indices whose predicted confidence $\hat{p}_i$ falls into the interval $I_m = (\frac{m-1}{M}, \frac{m}{M}]$. Then, the empirical accuracy is

$$\mathrm{acc}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} 1_{\hat{y}_i = y_i}.$$

Similarly, we define the *empirical confidence* as

$$\mathrm{conf}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} \hat{p}_i.$$

This leads to the natural definition of the *empirical ECE* as

$$\hat{\mathrm{ECE}} = \sum_{m=1}^{M} \frac{|B_m|}{n} |\mathrm{acc}(B_m) - \mathrm{conf}(B_m)|.$$

You can find our implementation of the empirical ECE in `util.py`. For more details, we refer you to Guo et al. (2015) (http://proceedings.mlr.press/v70/guo17a/guo17a.pdf).

We use the empirical ECE with $M = 20$ bins to measure your BNN's calibration. Since your model only outputs per-class confidences, the ECE always treats ambiguous samples as predicted wrongly. Whenever your ECE is larger than 0.1, we add the excess error to the cost described below as a penalty.

For the prediction cost, we use -1 as the label for ambiguous test samples and predicting "don't know". The prediction cost of $\hat{y} \in \{-1, 0, 1, 2, 3, 4, 5\}$ for a true label $y \in \{-1, 0, 1, 2, 3, 4, 5\}$ is

$$\ell(y, \hat{y}) = \begin{cases} 1, & \text{if } \hat{y} = -1 \\ 3, & \text{if } \hat{y} \neq -1 \text{ and } \hat{y} \neq y \\ 0, & \text{if } \hat{y} \neq -1 \text{ and } \hat{y} = y. \end{cases}$$

In words, predicting "don't know" always incurs a constant cost. Predicting the correct label for a non-ambiguous sample yields no cost, but predicting the wrong label or *any* label for an ambiguous sample incurs a larger cost.

This cost has a practical motivation: Consider the task of automatically classifying a country's land usage as a basis for policymaking. Predicting "don't know" requires a human to manually check the corresponding image, thereby incurring a fixed cost, even for ambiguous images. However, wrongly predicting a land usage type skews the basis for decision-making and thereby leads to potentially bad policies—resulting in a larger long-term cost.

To summarize, the overall cost is

$$\text{cost} = \text{mean prediction cost} + \max\{\hat{\text{ECE}} - 0.1, 0\}.$$

We calculate ECEs and the overall cost on the public and private test set individually. The checker will reveal both quantities for the public test set only.

HANDOUT FILES

The handout contains the following files:

- `solution.py` : solution template; put your entire solution in here
- `util.py` : additional helper functionality; do not change
- `runner.sh` : script to run the checker on your solution and generate a submission file
- `euler-guide.md` : guide that describes how to run this task on the euler cluster
- `env.yml` , `requirements.txt` : list of dependencies for the checker and euler, respectively; keep those in sync
- `train_xs.npz` , `train_ys.npz` : training data
- `val_xs.npz` , `val_ys.npz` : validation data
- `test_xs.npz` , `test_ys.bytes` : test data; labels are encrypted
- `Dockerfile` , `checker_client.py` , `pytransform` : technical files to run the checker
- `map_weights.pt` : MAP weights that serve as an initialization in SWAG

SUBMISSION WORKFLOW

1. Install and start Docker (https://www.docker.com/get-started). Understanding how Docker works and how to use it is beyond the scope of the project. Nevertheless, if you are interested, you could read about Docker's use cases (https://www.docker.com/use-cases).
2. Download handout (/static/task2_handout_e14a688d.zip)
3. The handout contains the solution template `solution.py` . Sections for possible implementations are marked with `# TODO` . Please make sure that your implementation preserves the names and signatures of all existing methods and classes. However, you are free to introduce new methods and attributes. Note: The `main()` method in the solution template is *for illustrative purposes only* and *completely ignored* by the checker!
4. You should use Python 3.8. You are free to use any other libraries that are not already imported in the solution template. Important: please make sure that you list all additional libraries together with their versions in the `requirements.txt` file provided in the handout.
5. Once you have implemented your solution, run the checker in Docker:
    - On Linux, run `bash runner.sh` . In some cases, you might need to enable Docker for your user (https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-user) if you see a Docker permission denied error.
    - On MacOS, run `bash runner.sh` . Docker might by default restrict how much memory your solution may use. Running over the memory limit will result in docker writing "Killed" to the terminal. If you encounter out-of-memory issues you can increase the limits as described in the Docker Desktop for Mac user manual (https://docs.docker.com/desktop/mac/). You could alternatively run your solutions on the ETH euler cluster. Please follow the guide specified by *euler-guide.md* in the handout.
    - On Windows, open a PowerShell, change the directory to the handout folder, and run `docker build --tag task2 .; docker run --rm -u $(id -u):$(id -g) -v "$(pwd):/results" task2` .
6. If the checker fails, it will display an appropriate error message. If the checker runs successfully, it will show your PUBLIC cost and ECE, tell you whether your solution passes the task, and generate a `results_check.byte` file. The `results_check.byte` file constitutes your submission and needs to be uploaded to the project server along with the code and text description in order to pass the project.
7. Your solution's cost determines your position on the leaderboard as well as whether you pass/fail this task. You pass if your solution's cost on the test data is at most the cost of our baseline on the test data.
8. We limit submissions to the server to 40 per team, with at most 20 in a 24 hour period.

EXTENDED EVALUATION

This part of the task is optional but highly encouraged. If you set the global variable `EXTENDED_EVALUATION` in the solution script to `True` , the checker generates additional plots from the validation set: samples that your model is most and least confident about, as well as a reliability diagram. All plots are stored as PDFs in your solution's directory.

GRADING

We provide you with **one test set** for which you have to compute predictions (and any other quantities we ask for, like uncertainty quantification). We have partitioned this test set into two parts (of the same size) and use it to compute a *public* and a *private* score for each submission. You only receive feedback about your performance on the public part in the form of the public score, while the private leaderboard remains secret. The purpose of this division is to prevent overfitting to the public score. Your model should generalize well to the private part of the test set. When handing in the task, you need to select which of your submissions will get graded and provide a short description of your approach. This has to be done **individually by each member** of the team. We will then compare your selected submission to our baseline. Your final grade for this task is the mean of the grades of your public and private scores. We will publish 2 baselines: easy and hard that each has a public and private score. Beating the easy baseline's public/private score guarantees you a 4.0 public/private grade respectively, while beating the hard baseline's public/private score guarantees you a 6.0 public/private grade respectively. Failing to beat the easy baseline's public/private score grants you a 2.0 public/private grade. For scores better than the easy baseline but worse than the hard baseline, we will grant a grade between 4.0 to 6.0 depending on how close you are to the hard vs easy baseline. We emphasise that this will likely not be a simple linear interpolation between the hard and easy baselines. In addition, for the pass/fail decision, we consider the code and the description of your solution that you submitted. That is, your code should be runnable and reproduce your predictions (see faq) and you should include a short writeup of your submitted solution. We emphasize that the public score leaderboard is just for fun: the scores of other teams will not effect the baseline or your own grade.

⚠ Make sure that you properly hand in the task, otherwise you may obtain zero points for this task.

FREQUENTLY ASKED QUESTIONS

⦿ WHICH PROGRAMMING LANGUAGE AM I SUPPOSED TO USE? WHAT TOOLS AM I ALLOWED TO USE?

You are free to choose any programming language and use any software library. However, **we strongly encourage you to use Python**. You can use publicly available code that was not produced directly for the purposes of this course, but you should specify the source as a comment in your code.

⦿ AM I ALLOWED TO USE MODELS THAT WERE NOT TAUGHT IN THE CLASS?

Yes. Nevertheless, the baselines were designed to be solvable based on the material mentioned in the project description or taught in the class up to the second week of each task.

⦿ IN WHAT FORMAT SHOULD I SUBMIT THE CODE?

You can submit it as a single file (main.py, etc.; you can compress multiple files into a .zip) having max. size of 1 MB. If you submit a zip, please make sure to name your main file as *main.py* (possibly with other extension corresponding to your chosen programming language).

⦿ WILL YOU CHECK / RUN MY CODE?

We will check your code and compare it with other submissions. We also reserve the right to run your code. Please make sure that your code is runnable and your predictions are reproducible (fix the random seeds, etc.). Provide a readme if necessary (e.g., for installing additional libraries).

⦿ SHOULD I INCLUDE THE DATA IN THE SUBMISSION?

No. You can assume the data will be available under the path that you specify in your code.

⦿ CAN YOU HELP ME SOLVE THE TASK? CAN YOU GIVE ME A HINT?

As the tasks are a graded part of the class, **we cannot help you solve them**. However, feel free to ask general questions about the course material during or after the exercise sessions.

⦿ CAN YOU GIVE ME A DEADLINE EXTENSION?

⚠ We do not grant any deadline extensions!

⦿ CAN I POST ON MOODLE AS SOON AS HAVE A QUESTION?

This is highly discouraged. Remember that collaboration with other teams is prohibited. Instead,

- Read the details of the task thoroughly.
- Review the frequently asked questions.
- If there is another team that solved the task, spend more time thinking.
- Discuss it with your team-mates.

⦿ WHEN WILL I RECEIVE THE PRIVATE SCORES? AND THE PROJECT GRADES?

We will publish the private scores, and corresponding grades before the exam at the latest.