...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 neural networks into more high-stakes domains , such as healthcare and self-driving vehicles, neural networks must be able to report the 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, BNN's 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) is to estimate weights through Maximum a Posteriori estimation

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

where x_i is the training input and y_i is the training label. However, since the posterior and predictions are intractable, in practice we need approximate inference techniques. In this task we provide four options for implementating Bayesian neural networks: *MC Dropout (https://arxiv.org/pdf/1506.02142.pdf), Ensemble Learning (https://proceedings.neurips.cc/paper/2017/file/9ef2ed4b7fd2c810847ffa5fa85bce38-Paper.pdf), MCMC (https://www.stats.ox.ac.uk/~teh/research/compstats/WelTeh2011a.pdf) and Bayes Backprop (http://proceedings.mlr.press/v37/blundell15.pdf)* A reasonable implementation of any single one of the four methods can pass the task. We recommend starting with the one you find most interesting. If you have time, you can complete multiple approaches to better understand the course content. Even if you already understand how the algorithm works, we recommend reading the corresponding paper we linked to above for any method you choose to implement.

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, $\mathcal{X} = \mathbb{R}^{784}$ are flattened images and $\mathcal{Y} = \{0, \dots, 9\}$ 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.

To understand why uncertainty calibration for predictive models is important, 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

YOUR TASK

GENERAL

Your task is to implement a Bayesian Neural Network for multi-class classification using one or more of the following methods. You have to solve several concretely defined subtasks to implement a chosen method.

All tasks should be implemented by filling-in the solution template `solution.py`, which can be found in the handout. You can get a better understanding of the template components by looking at the `Framework` and `DummyTrainer` classes.

Regardless of which method you choose to use, you will need to complete the following tasks:

1. Choose the method you want to implement in `run_solution`. You can also build your own approach but it is highly recommend to first pass the baseline with approaches that we provide.
2. Modify the network structure if necessary in class `MNISTNet`. You can change the depth or width of the model.
3. If you choose to design your own architecture from scratch, implement your own model in class `SelfMadeNetwork`.

You can find the relevant places in the solution template for the above tasks with the keyword `TODO General` Please note that you likely do not need to change the model architecture significantly in order to pass the baseline, but making some improvements may make passing the baseline easier. We'd suggest starting with simply modifying the width and depth of `MNISTNet`. The emphasis of the task is on designing BNNs, not hyperparameter tuning a model architecture.
We do not provide suggested hyperparameters for each method. While you may need to do some hyperparameter tuning, you should not need to do extensive hyperparameter tuning since a range of reasonable hyperparameters should allow each method to pass the task. To do some simple manual hyperparameter tuning, we recommend your team make an excel to record the method used, ece (see metrics section later), accuracy and compound score every time you run `runner.sh`. While implementing a method you may need to introduce additional hyperparameters beyond those that have been specified for you in the solution template.

After you pass the baseline you might want to try and improve your score further. For each method, we provide some potential directions that you can work on in the method section and general methods extension section. Note that there is no guarantee that these advanced tasks will improve your score and you are encouraged to try new directions or create new methods yourself.

MONTE CARLO DROPOUT

This method should be implemented in the class `DropoutTrainer`. Your concrete tasks are as follows:

1. Initialize the network and optimizer in `DropoutTrainer`. You can choose the same

initialization as `DummyTrainer` , though other choices could be used too.

2. Implement the MC Dropout Training process and calculate the loss. You can implement the loss function from the corresponding literature.
3. Implement the MC Dropout prediction. You need to sample from your trained model multiple times for Monte Carlo integration.
4. Run experiments and tune hyperparameters. You may choose to introduce addditional hyperparameters beyond the ones already in the template.

You can search all `TODO` s by keyword `TODO: MC_Dropout` . The MC Dropout method will get different score each time, due to randomness in the algorithm, so we recommend using the same seed for better reproducibility. `torch.manual_seed(0)` . If you have more time and want to improve the performance of Monte Carlo Dropout, you might consider the general methods extension section further down.

DEEP ENSEMBLE

This method should be implemented in the class `DeepEnsemble` . Your concrete tasks are as follows:

1. Initialize the network and optimizer in `EnsembleTrainer` . You can choose the same initialization as `DummyTrainer` , though other choices could be used too.
2. Implement the Ensemble Training process and calculate the loss. You can implement the loss function from the corresponding literature.
3. Implement the Ensemble prediction. You need to combine the outputs from different ensemble members.
4. Run experiments and tune hyperparameters. You need to decide how large your ensemble will be.

You can search all `TODO` s by keyword `TODO: Ensemble` . If you have more time and want to improve the performance of the Deep Ensemble, you might consider the following first and also the suggestions in the general extensions section:

- Implement bootstrap sampling. You can sample different bootstrapped datasets for each member in the ensemble.

For implementing the bootstrap sampling you may need to implement machinery for sampling datapoints uniformly, with replacement, from the training data.

STOCHASTIC GRADIENT LANGEVIN DYNAMICS

We provide the SGLD optimizer in `utils.SGLD` . Compared to the work Bayesian Learning via Stochastic Gradient Langevin Dynamics (https://www.stats.ox.ac.uk/~teh/research/compstats/WelTeh2011a.pdf) we decrease the scale of the noise for stability. This method should be implemented in the class `SGLDTrainer` . Your concrete tasks are as follows:

1. Initialize the network and `SGLDSequence` object. This sequence contains a history of SGLD networks. During training, the SGLD method produces a sequence of weights. However, it is expensive to store all of them in our sequence, so normally we will sample them only after a "burn-in" period
2. Implement the SGLD Training process and calculate the loss. You can implement the loss function from the corresponding literature.
3. Implement the addition and deletion of models to the SGLD model sequence. To avoid memory explosion we fix the maximum size of the sequence of samples. You will save a "snapshot" at regular fixed intervals. When the sequence exceeds the maximum length, you will have to delete the oldest snapshot. You won't need to add any samples during the burn-in period.
4. Implement the SGLD prediction by combining the outputs of the sequence members.
5. Run experiments and tune hyperparameters. You need to decide the burn-in time and snapshot sampling interval.

You can search all `TODO` s by keyword `TODO: SGLD`. We recommend you use the deque (https://docs.python.org/3/library/collections.html#collections.deque) for storing SGLD models. If you have more time and want to improve the performance of SGLD, you might consider the following and also the suggestions in the general extensions section:

- Learning rate annealing.

So far we fixed the learning rate, however it might not converge to the true posterior over the weights. The learning rate can be annealed according to the Robbins-Monro conditions. Details of this can be found in this paper (https://www.stats.ox.ac.uk/~teh/research/compstats/WelTeh2011a.pdf).

BAYES BY BACKPROP

This method should be implemented in the class `BackpropTrainer`. Your concrete tasks are as follows:

1. Create a suitable prior and variational posterior in the class `BayesianLayer` that represents the Bayesian equivalent of an affine neural network layer.
2. Perform a forward pass as described in this method's docstring from the class `BayesianLayer`.
3. Implement the class `BayesNet` that represents a full BNN.
4. Complete the `log_likelihood` function and `sample` function of Univariate Gaussian
5. Complete the `log_likelihood` function and `sample` function of Multivariate Diagonal Gaussian
6. Implement the class `BackpropTrainer` to perform Bayes by backprop and use the resulting network for inference.
7. Run experiments and tune hyperparameters. You need to decide the network depth and width.

You can search all `TODO` s by keyword `TODO: Backprop`. The template contains an abstract class `ParameterDistribution` that models arbitrary distributions over weights (both priors and posteriors). We already provide the structure for univariate and diagonal multivariate Gaussians in the classes `UnivariateGaussian` and `MultivariateDiagonalGaussian` respectively. After passing the baseline, you might want to investigate varying the prior parameters or adding more complex prior distributions that yield better uncertainty estimates.

GENERAL METHODS EXTENSION

If you have more time and want to improve the results of methods MC Dropout, Ensemble learning and SGLD, you can try following suggestions.

- Determine and implement suitable priors for the network weights. Usually one would use a zero-mean Gaussian distribution to model the network priors, but alternative distributions might be more successful.
- Changing from a homoscedastic noise model to a heteroscedastic noise model. See below for more information on the difference between these models.

Heteroscedastic uncertainty is uncertainty in a datapoint's label that changes depending on the datapoint's features. Image classification is undoubtedly heteroscedastic because for different inputs the aleatoric uncertainty will be different. For example, consider sharp images and blurry images. Work "What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?" (https://arxiv.org/abs/1703.04977) shows that a heteroscedastic noise model can improve performance on classification tasks. Implementing a heteroscedasrtic noise model like the one described in this paper will take a lot of time, so we strongly recommend that you pass the baseline first, and then consider it if you are interested.

We also encourage you to read recent literature and implement your own extension.

COMBINE DIFFERENT METHODS

Please note that this section is optional and you should try combining methods only after passing the baseline. We provide you with an interface where you can train a list of models seperately in the function `run_solution`. Then, you can write your own rule to combine the prediction in function `combined_prediction`

In this extension the concrete tasks are as follows:

1. Set the `combined_model` to `True`. It will then train and evalute a list of models.
2. Construct the combined model list in `run_solution`. We recommend that you pass the baseline with each method you use first.
3. Implement the function `combined_prediction`. You need to decide on a rule for how you want to combine the predictions of different methods. See Wikipedia (https://en.wikipedia.org/wiki/Ensemble_averaging_(machine_learning)) for more information about ensembling trained models.

You can search all `TODO` s by keyword `TODO: Combined model`. Moreover, during implementation we recommend to set `EXTENDED_EVALUATION` to `False` because it will run extend evalution multiple times which is time-consuming and will only save the extended evaluation results of last model.

GENERAL ADVICE

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). While you are technically free to use a different library, you will have to re-implement a significant amount of code. We will only provide support for PyTorch-related technical issues.

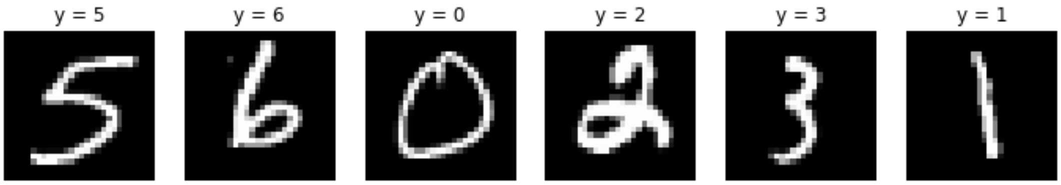
If you are failing to pass the baseline and suspect it could be due to poor hyperparameter selection, the literature link we provided for each method earlier in the description might provide some useful direction.

DATASET

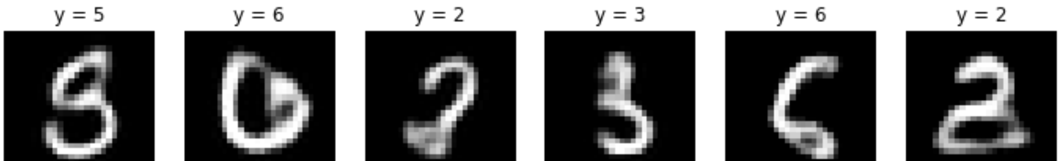
One of the most famous datasets in machine learning is the MNIST dataset (LeCun et al., 1998) (http://yann.lecun.com/exdb/mnist/). MNIST is a large database of handwritten digits which is commonly used for training various image processing systems. The MNIST dataset contains images consisting of 28x28 grayscale pixels with a label in the range $\{0, \ldots, 9\}$.

Our training set consists of the original 20000 MNIST training images. For the test set, we use a modified version of MNIST. The test images exhibit varying degrees of ambiguity and are further rotated by random angles. Ambiguity introduces aleatoric uncertainty since a true image label might not be uniquely identifiable. The rotations introduce epistemic uncertainty since your network only observes non-rotated images during training. Note that this is different from many typically studied settings in machine learning where the train and test data come from the same distribution.

The following are examples of training set images:



The following are examples of test set images:



We have reshaped all images (in both training and test sets) into 784-dimensional vectors for you. Therefore, even though the original data is image data, we suggest that you use feedforward neural network architectures rather than convolutional architectures.

You pass the task if your solution's PUBLIC ECE and accuracy (both measured on the test set) satisfy the constraints described below.

METRICS

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 $M = 30$ bins to measure your BNN's calibration. Since we also want your BNN to predict well on average, we further measure your model's accuracy. You pass this task if you match our baseline *both* in terms of empirical ECE and accuracy. Concretely, you pass if your model's metrics on the PUBLIC test set satisfy

ECE	**Accuracy**
<= 0.05	>= 0.65

Note that, while 65% accuracy seems low, the large amounts of uncertainty in the test set will make it difficult to reach significantly higher accuracy levels.

Lastly, we use a compound score defined as

$$\text{score} = \text{accuracy} + 3 * \left(\frac{1}{2} - \mathrm{E\hat{C}E}\right).$$

This score is *irrelevant for passing* and only determines your position on the leaderboard. The checker will reveal your compound score on the test set. If you submit a solution to the website that does not pass, the leaderboard will show that solution to have a score of 0 instead of a value corresponding to the above formula. ...