Beyond BBB: Practical Alternatives to Posterior Approximation in Bayesian Neural Networks

Tomasz Kuśmierczyk

February 18, 2025

This research is part of the project No. ***2022/45/P/ST6/02969*** *co-funded by the National Science Centre and the European Union Framework Programme for Research and Innovation Horizon 2020 under the Marie Skłodowska-Curie grant agreement No. 945339. For the purpose of Open Access, the author has applied a CC-BY public copyright licence to any Author Accepted Manuscript (AAM) version arising from this submission;*

Bayesian Learning in Neural Networks

Model Parameters as Random Variables:

- In a neural network (NN), we have weight parameters $\theta = \{w_1, w_2, \ldots, w_m\}$ (possibly millions).
- Bayesian approach: place a prior on θ and condition on data D to get the posterior $p(\theta \mid D)$.
- Posterior following Bayes rule:

$$p(\theta \mid D) \propto p(D \mid \theta)\, p(\theta)$$

- Predictive distribution integrates over θ:

$$p(\mathbf{y}^* \mid \mathbf{x}^*, D) \;=\; \int p(\mathbf{y}^* \mid \mathbf{x}^*, \theta)\, p(\theta \mid D)\, d\theta.$$

Bayesian Neural Networks: Quick Recap

- In practice, exact posterior inference for modern deep networks is intractable.
- **Aim:** Approximate $p(\theta \mid \mathcal{D})$ to quantify predictive uncertainty.
- Common approximations:
  - Variational Inference (including **BBB**).
  - **Stochastic Weight Averaging Gaussian (SWAG)**.
  - **Laplace Approximation**.
  - MCMC-based methods, . . .

Bayesian vs MAP

$$\mathcal{L}(\theta \mid \mathcal{D}) = CE(\mathcal{D}|\theta) + \text{weight-decay}(\frac{1}{2\sigma^2})$$

Bayesian vs MAP

$$\mathcal{L}(\theta \mid \mathcal{D}) = CE(\mathcal{D}|\theta) + \text{weight-decay}(\frac{1}{2\sigma^2})$$

Say $\mathcal{D} = \{(x, y)\}$, $p(y \mid \theta, x) = Cat(nn(x))$ and $p(\theta) = \mathcal{N}(\theta|0, \sigma^2 I)$
Then:

$$\begin{aligned}\mathcal{L}(\theta \mid \mathcal{D}) &= -\log p(\mathcal{D} \mid \theta) - \log p(\theta) \\ &= -\log \left(p(\mathcal{D} \mid \theta)\, p(\theta)\right)\end{aligned}$$

Bayesian vs MAP

$$\mathcal{L}(\theta \mid \mathcal{D}) = CE(\mathcal{D}|\theta) + \text{weight-decay}(\frac{1}{2\sigma^2})$$

Say $\mathcal{D} = \{(x, y)\}$, $p(y \mid \theta, x) = Cat(nn(x))$ and $p(\theta) = \mathcal{N}(\theta|0, \sigma^2 I)$
Then:

$$\begin{aligned}\mathcal{L}(\theta \mid \mathcal{D}) &= -\log p(\mathcal{D} \mid \theta) - \log p(\theta) \\ &= -\log\left(p(\mathcal{D} \mid \theta)\, p(\theta)\right)\end{aligned}$$

vs.

$$\log p(\theta \mid \mathcal{D}) \propto \log(p(\mathcal{D} \mid \theta)\, p(\theta))$$

$\rightarrow$ standard optimization with reqularization finds max of the posterior: $\theta_{\text{MAP}} = \text{argmax} p(\theta|\mathcal{D})$

Approximate methods: general framework

Goal: Approximate $p(\theta \mid \mathcal{D}) \approx \mathcal{N}(\hat{\mu}, \hat{\Sigma})$
Steps:

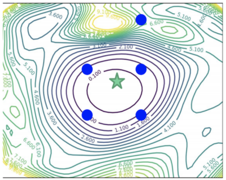
- Gradient-based optimization (with specific scheduler and/or regularization)
- Postprocessing to get $\hat{\mu}, \hat{\Sigma}$

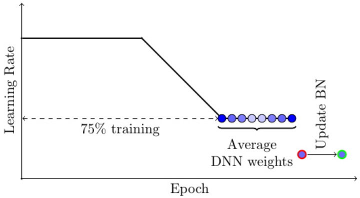
Challenges in Approximate Posterior Learning

- **Approximation quality:** Trade-off between computational feasibility and fidelity to the true posterior.
- **Local minima and multi-modal posteriors:** Loss surfaces can be highly non-convex.
- **Implementation complexities:**
    - Additional overhead for sampling or for second-order information (e.g., Hessians).

SWA: High-Level Intuition

- Neural networks often converge to "good" solutions near the end of training.
- By saving multiple *snapshots* of these parameters, we empirically estimate a distribution.





Source: `https://pytorch.org/blog/stochastic-weight-averaging-in-pytorch/`

SWAG: Steps

1. Train your model as usual
2. After N epochs fix your optimization scheduler and periodically save/collect model parameters.
    - *Compute running mean* of these parameters to get μ_{SWAG}.
    - *Compute second moment* or low-rank approximations to get covariance.
3. *Form a Gaussian* $\mathcal{N}(\mu, \Sigma)$ to approximate posterior.
4. *Sample* from $\mathcal{N}(\mu, \Sigma)$ for predictive uncertainties.

SWAG: Key Equations

Posterior Approximation:

$$p(\theta|\mathcal{D}) = \mathcal{N}\big(\hat{\mu},\ \frac{1}{2}(\mathrm{diag}(\hat{v}) + \frac{1}{K-1}\underbrace{D\,D^\top}_{\text{low-rank}})\big),$$

where

$$\hat{\mu} = \frac{1}{T}\sum_{i=1}^{T}\theta_i,$$

$$\hat{v} = \frac{1}{T}\sum_{i=1}^{T}\theta_i^2\ -\ \hat{\mu}^2,$$

$$\Sigma_{\text{low-rank}} \approx \frac{1}{K-1}\sum_{i}^{K} D_i D_i^T, \quad D_i = \theta_i - \hat{\mu}$$

D is a queue of last K deviations from the mean $\rightarrow$
$\Sigma_{\text{low-rank}}$ is estimated from the last K sets of parameters and has rank K

SWAG: Running Means and Variances

```
def collect_model(self, base_model):
    ...
    # update SWAG means & sq. means
    for (module, name), base_param in zip(self.params, base_model_params):
        mean_ = module.__getattr__(f"{name}_mean")
        sq_mean_ = module.__getattr__(f"{name}_sq_mean")

        mean_ = mean_*self.n_models.item()/(self.n_models.item()+1.0) \
                + base_param.data/(self.n_models.item()+1.0)
        sq_mean_ = sq_mean_*self.n_models.item()/(self.n_models.item()+1.0)\
                + base_param.data**2/(self.n_models.item()+1.0)

        module.__setattr__(f"{name}_mean", mean_)
        module.__setattr__(f"{name}_sq_mean", sq_mean_)
    self.n_models.add_(1)
```

SWAG: Running Covariance

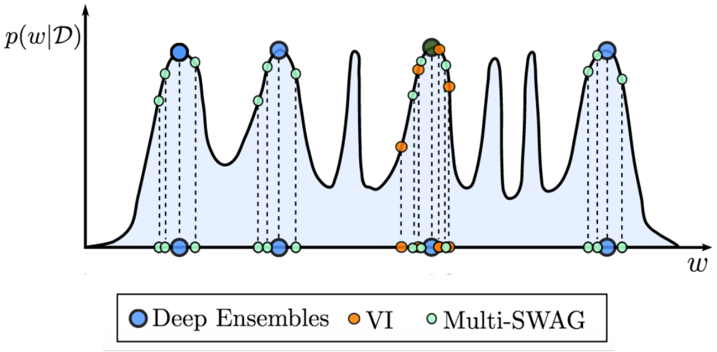
```
# square root of covariance matrix
if self.no_cov_mat is False:
    cov_mat_sqrt = module.__getattr__("%s_cov_mat_sqrt" % name)

    # block covariance matrices, store deviation from current mean
    dev = (base_param.data - mean).view(-1, 1)
    cov_mat_sqrt = torch.cat((cov_mat_sqrt, dev.view(-1, 1).t()), dim=0)

    # remove first column if we have stored too many models
    if (self.n_models.item() + 1) > self.max_num_models:
        cov_mat_sqrt = cov_mat_sqrt[1:, :]
    module.__setattr__("%s_cov_mat_sqrt" % name, cov_mat_sqrt)
```

SWAG: Sampling

$$\tilde{\theta} = \hat{\mu} + \frac{1}{\sqrt{2}}\mathrm{diag}(\sqrt{\hat{v}})\,\epsilon_1 + \frac{1}{\sqrt{2(K-1)}}D\,\epsilon_2,$$

with $\epsilon_1 \sim \mathcal{N}(0, I_d)$, $\epsilon_2 \sim \mathcal{N}(0, I_K)$.

Multi-SWAG



Source: `https://arxiv.org/pdf/2002.08791`

Laplace Approximation: Overview

- **Idea:** Approximates $p(\theta \mid \mathcal{D})$ by a Gaussian centered at the MAP solution:

$$p(\theta \mid \mathcal{D}) \approx \mathcal{N}\big(\theta_{\mathrm{MAP}},\, H^{-1}\big),$$

  where H is (an approximation to) the Hessian of the negative log posterior at θ_{MAP}.
- H is set to $\nabla_\theta^2[-\log p(\mathcal{D} \mid \theta) - \log p(\theta)]\big|_{\theta_{\mathrm{MAP}}}$.
- **Variants**: Diagonal, Full, Kronecker-factored (kron), Low-rank approximations, etc.

Laplace Approximation: Steps

Idea:

- Perform standard training to get $\theta_{\text{MAP}} \approx \arg\max p(\theta \mid D)$.
- Approx. the local posterior by a Gaussian with covariance from the Hessian or a variant.

Steps:

1. *Train model* to get θ_{MAP}.
2. *Compute Hessian approximation* $H \approx \nabla_\theta^2 \mathcal{L}(\theta_{\text{MAP}})$.
3. *Invert* (approximately) $\Sigma \approx -H^{-1}$.
4. *Sample* θ from $\mathcal{N}(\theta_{\text{MAP}}, \Sigma)$ or do linearized predictive (helps a lot with GGN!).

Laplace Redux

```
la = Laplace(model, # pre-trained model
           'classification', # likelihood -> CE
           prior_precision=1., # prior params
           hessian_structure='kron' # covariance approximation
           subset_of_weights='all', # on which weights
           )
la.fit(train_dataloader)
```

Laplace: Collecting Curvature Information

Excerpt from baselaplace.py *where curvature is accumulated:*

```python
class BaseLaplace:
    def __init__(self, model, likelihood, sigma_noise=1., prior_precision=
         None, ...):
        self.model = model
        self.likelihood = likelihood
        # Initialize Hessian or curvature approx...

    def fit(self, train_loader):
        # For each batch, gather curvature information (e.g. GGN, Hessian):
        for batch in train_loader:
            self.model.zero_grad()
            loss_batch, H_batch, f = self._curv_closure(batch, N)
            self.loss += loss_batch
            self.H += H_batch
        self.n_data += N
```

Note: _curv_closure is the function that computes Hessian approximations for each batch (or the GGN, etc.).

Generalized Gauss-Newton (GGN) Approximation

- **GGN Approximation (Generalized Gauss-Newton):** replaces the exact Hessian with
$$H_{\mathrm{GGN}} \approx J^{\top} \left(\nabla_f^2 \ell(f; y)\right) J,$$
where
    - J is the Jacobian of the network outputs w.r.t. parameters,
    - $\nabla_f^2 \ell$ is the Hessian of the neg-likelihood w.r.t. the model outputs (often simpler to compute).
- **Key Advantage:** avoids computing the second derivatives of each network layer directly, using backprop for Jacobian-vector products instead.

Kronecker Factorization of the Hessian

- **Motivation:** Computing and storing the full Hessian for a large network is infeasible (costly in both memory and computation).
- **Key Idea:** Approximate the layer-wise Hessian as a Kronecker product of smaller matrices. For a layer with parameter shape $(d_{\text{out}} \times d_{\text{in}})$, the Hessian can be approximated as:

  $$H \approx A \otimes B,$$

  where $A \in \mathbb{R}^{d_{\text{out}} \times d_{\text{out}}}$ and $B \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}$ capture output- and input-side curvature, respectively.
- **Why This Helps:**
  - Inversion of H is reduced to inverting A and B individually:

    $$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}.$$

  - *Memory Savings:* Instead of storing a full $(d_{\text{out}} d_{\text{in}}) \times (d_{\text{out}} d_{\text{in}})$ matrix, only two much smaller $(d_{\text{out}} \times d_{\text{out}})$ and $(d_{\text{in}} \times d_{\text{in}})$ matrices are needed.
  - *Computation Benefits:* Determinants and matrix products factorize accordingly, improving efficiency for posterior covariance computations.

Laplace Approximation: Model Linearization

$$p(\theta|D) \approx q(\theta) = \mathcal{N}\left(\theta_{\mathrm{MAP}}, H^{-1}\right).$$

BNN predictive (Eq. (9)) GLM predictive (Eq. (13))

$$\begin{array}{c} p_{\mathrm{BNN}}(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \\ \int q(\boldsymbol{\theta}) p(\mathbf{y}|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}))\, \mathrm{d}\boldsymbol{\theta} \end{array} \xrightarrow{\mathrm{GGN}} \begin{array}{c} p_{\mathrm{GLM}}(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \\ \int q(\boldsymbol{\theta}) p(\mathbf{y}|\mathbf{f}_{\mathrm{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta}))\, \mathrm{d}\boldsymbol{\theta} \end{array}$$

$$\mathbf{f}_{\mathrm{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{f}(\mathbf{x}, \boldsymbol{\theta}^*) + \nabla_{\boldsymbol{\theta}} \mathbf{f}(\mathbf{x}, \boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*}^{\top} (\boldsymbol{\theta} - \boldsymbol{\theta}^*)$$

Source: `https://arxiv.org/pdf/2008.08400`

Priors

- With Gaussian posteriors Gaussian priors are typically used
- In SWAG just as weight decay during optimization
- In Laplace as weight decay during optimization and when computing Hessian
  - Can be fit by optimizing Marginal Log-likelihood

SWAG vs. Laplace

SWAG

- *Snapshots approach*: Takes advantage of final training fluctuations.
- *Pros*: Easy to implement, minimal overhead, good in practice if final epochs explore parameter space sufficiently.
- *Cons*: Might not capture full curvature; depends on snapshot frequency/phase.

Laplace

- *Hessian-based approach*: Local Gaussian near θ_{MAP}.
- *Pros*: Classic, interpretable in terms of second-order expansions, can incorporate advanced factorization.
- *Cons*: Hessian computations can be costly for large nets unless further approximations (diag/K-FAC/low-rank).

References

- G. Maddox et al. (2019): *Simple and Principled Bayesian Inference with SWAG*. A. Wilson et. al (2023): *Bayesian Deep Learning and a Probabilistic Perspective of Generalization*
- D. MacKay (1992): *Bayesian Methods for Adaptive Models*.
- A. Immer et al. (2021): *Improving predictions of Bayesian neural nets via local linearization*
- E. Daxberger et al. (2021): *Laplace Redux – Effortless Bayesian Deep Learning*
- A. Ritter et al. (2018): *Scalable Laplace Approximations for Neural Networks (K-FAC).*