Natural-Gradient Variational Inference for Bayesian Convolutional Neural Networks

Anirudh Jain and Mohammad Emtiyaz Khan
RIKEN AIP

Abstract

We present a fast and scalable way for Bayesian training of deep learning models which can be implemented within the Adam optimizer. Our algorithm runs out of the box without requiring any modifications to the model architecture. We show examples of models with Linear and Convolutional layers such as LeNet5. We also support minibatching and pooling layers.

1 Introduction

There have been several attempts to compute uncertainty in deep neural networks by using Approximate Bayesian Inference. These uncertainty estimates help us to understand the outputs of black box systems such as neural networks. However, existing approaches require the user to define a distribution over the weights. In the Gaussian case, this leads to twice the number of parameters compared to a non-Bayesian model [Blundell et al., 2015].

The natural gradients approach simplifies this problem [Khan et al., 2018] but it requires computation of the square of individual gradients in a minibatch. See line 6 in Algorithm 1 which shows the pseudocode for the Variational Online Gauss-Newton algorithm proposed in [Khan et al., 2018]. This is difficult to implement in existing codebases as they are optimized to directly return the sum of gradients over the minibatch. To implement this step, extra effort is required in modifying the forward pass to store the pre-activations and inputs for each layer. This limits the implementation to fully connected layers and it is difficult to modify the forward pass for other networks such as Residual Networks [He et al., 2016].

In this note, we present a new implementation that enables the use of convolutional and max pooling layers. We use the compute graph in PyTorch shown in Figure 1 to avoid making modifications to the model. A forward hook can be used to obtain the inputs and a backward hook for the gradients w.r.t. pre-activations. This allows us to use deep learning models with convolutional and max-pooling layers such as LeNet5.

Require: α: Learning Rate
Require: N: Training Set Size
Require: λ: Prior Precision
Require: $\gamma \in [0, 1)$: Coefficient for running average of squared gradients
Require: $\log p(\mathbf{D}|\mathbf{w})$: Log-likelihood function
Require: $\boldsymbol{\mu}_0$: Initial Parameter Vector
$\mathbf{s}_o \leftarrow 0$ (Initialize Precision)
$t \leftarrow 0$ (Initialize timestep)

while not converged **do**
  $\mathbf{w} \leftarrow \boldsymbol{\mu}_t + \boldsymbol{\sigma}_t \circ \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$, $\boldsymbol{\sigma}_t \leftarrow 1/\sqrt{N\mathbf{s}_t + \lambda}$
  Randomly sample a minibatch $\mathcal{M}$,
  $\mathbf{g}_i \leftarrow -\nabla \log p(\mathbf{D}_i|\mathbf{w}), i \in \mathcal{M}$
  $\hat{\mathbf{g}} \leftarrow \sum_{i=1}^{M} \mathbf{g}_i / M$
  $\mathbf{s}_{t+1} \leftarrow \gamma \mathbf{s}_t + (1 - \gamma)(\sum_{i=1}^{M} \mathbf{g}_i^2 / M)$
  $\boldsymbol{\mu}_{t+1} \leftarrow \boldsymbol{\mu}_t - \alpha(\hat{\mathbf{g}} + \lambda \boldsymbol{\mu}_t / N)/(\mathbf{s}_t + \lambda / N)$
  $t \leftarrow t + 1$
end
Return : $\boldsymbol{\mu}_t$ (Returned Parameters)

Algorithm 1: Variational Online Gauss-Newton(VOGN) Algorithm

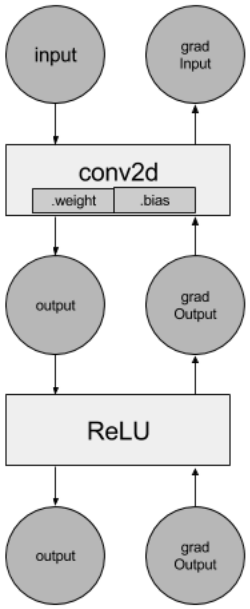
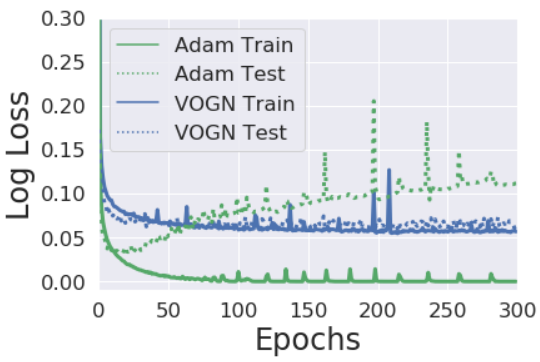


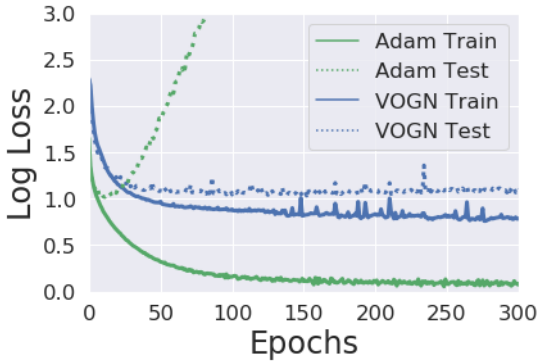
Figure 1: The compute graph in PyTorch. grad Output is the gradients w.r.t. pre-activations for the given layer. input and grad Output can directly be used within the optimizer using hooks to compute the individual gradients in a minibatch.

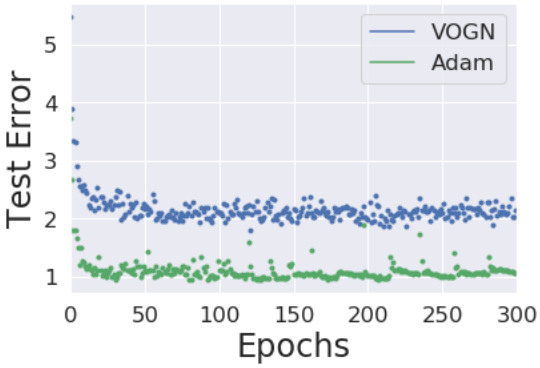
2 Empirical Evaluation

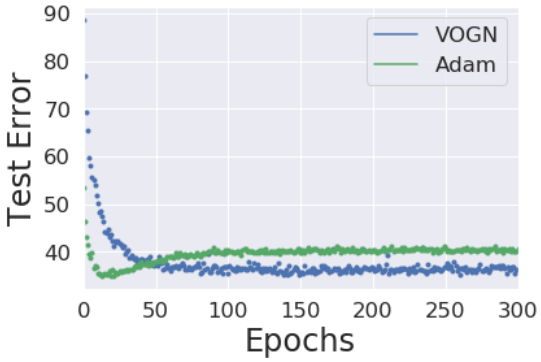
(a) MNIST



(b) CIFAR10







Test		
	VOGN	Adam
Log Loss	**0.065**	0.108
Error	2.109	**1.079**
Train		
	VOGN	Adam
Log Loss	0.058	**0.001**
Error	1.718	**0.026**

Test		
	VOGN	Adam
Log Loss	**1.130**	8.341
Error	**37.01**	40.47
Train		
	VOGN	Adam
Log Loss	0.815	**0.077**
Error	27.18	**2.248**

Figure 2: Evaluation metrics on Train and Test sets for both optimizers. Adam overfits while VOGN does a good job of keeping test and train errors close. VOGN outperforms Adam on CIFAR10 but underperforms on MNIST for test accuracy. For test log loss, VOGN is better than Adam in both cases. Model architectures given in Table 1.

CovNet for MNIST						
Layer No.	Layer Type	Width	Stride	Padding	Input Shape	Non-Linearity
1	Convolutional(5 x 5)	6	1	0	M x 3 x 28 x 28	ReLU
	MaxPool(2 x 2)		2	0	M x 6 x 24 x 24	
2	Convolutional(5 x 5)	16	1	0	M x 6 x 12 x 12	ReLU
	MaxPool(2 x 2)		2	0	M x 16 x 8 x 8	
3	Fully-connected	10			M x 256	SoftMax
LeNet5 for CIFAR10						
Layer No.	Layer Type	Width	Stride	Padding	Input Shape	Non-Linearity
1	Convolutional(5 x 5)	6	1	0	M x 3 x 32 x 32	ReLU
	MaxPool(2 x 2)		2	0	M x 6 x 28 x 28	
2	Convolutional(5 x 5)	16	1	0	M x 6 x 14 x 14	ReLU
	MaxPool(2 x 2)		2	0	M x 16 x 10 x 10	
3	Fully-connected	120			M x 400	ReLU
4	Fully-connected	84			M x 120	ReLU
5	Fully-connected	10			M x 84	SoftMax

Table 1: Model architectures used for the experiments (Here M is the batch size).

3 Details of calculating individual gradients in a minibatch for Convolutional Layers

The existing implementation of VOGN allows us to train simple feedforward networks as shown in Algorithm 1. As we can see in line in 6, we need to compute the individual gradients w.r.t. model parameters. The solution in [Goodfellow, 2015], used in the existing implementation provided by authors of [Khan et al., 2018], is limited to only fully connected layers and we extend it to convolutional layers.

Consider a convolutional filter $\mathbf{W}$ with dimensions **[k,k]** and inputs $\mathbf{X}$ with dimensions [M, C, H, W]. Here M is the the size of the minibatch, C is the number of channels, H,W are the spatial dimensions and k is the filter size. Assuming the stride to be 1 and 0 padding for our convolutions, the filter $\mathbf{W}$ will act on [k,k] patches of $\mathbf{X}$ shifting by 1 pixel sequentially.

Let [[.]] be an expansion operator such that,

$$[[X]]_{M,C,(W-k+1)h+w} = (X_{M,C,i,j})_{\substack{h\leq i\leq h+k \\ w\leq j\leq w+k}}, 0 \leq h < H \text{ and } 0 \leq w < W$$

$[[X]]$ is the input for the filter $\mathbf{W}$, $\mathbf{S}$ are pre-activations and $\mathbf{A}$ are the activations. We can compute the gradients and the square of gradients for a loss L as,

$$S_i = [[A_{i-1}]] \times W$$

$$G_i = \frac{\partial L}{\partial S_i}$$

$$\frac{\partial L}{\partial K} = [[A_{i-1}]] \times G_i$$

$$\left[\frac{\partial L}{\partial K}\right]^2 = ([[A_{i-1}]])^2 \times (G_i)^2$$

References

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*, 2015.

Ian Goodfellow. Efficient per-example gradient computations. *arXiv preprint arXiv:1510.01799*, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Mohammad Emtiyaz Khan, Didrik Nielsen, Voot Tangkaratt, Wu Lin, Yarin Gal, and Akash Srivastava. Fast and scalable bayesian deep learning by weight-perturbation in adam. *arXiv preprint arXiv:1806.04854*, 2018.