Robustness of Bayesian Approach to Gradient-Based Attacks

Sowmya Jayaram Iyer
Department of Computer Science
Purdue University
West Lafayette, IN 47907
`jayarami@purdue.edu`

Abstract

Despite significant efforts, Deep neural networks(DNNs) are vulnerable to adversarial examples, which are one of the chief hurdles to the adoption of deep learning in safety-critical applications. Numerous Adversarial defense strategies have been proposed in recent works, however, they typically show limited feasibility due to compromise on efficiency.Theoretical findings have shown that, in suitably defined large data limit, BNNs posteriors are robust to gradient-based adversarial attacks.Thus, this study aims to demonstrate the theoretical robustness of Bayesian neural architectures against multiple white-box attacks and list empirical findings from the same.

1 Introduction

Adversarial attacks have been extensively studied since Szegedy et al. 's seminal work [2], and even deep learning models trained on very large data sets are shown to be vulnerable to such attacks [3]. As a result, developing machine learning models that are resilient to adversarial perturbations is an essential precondition for their application in safety-critical scenarios. Vulnerability to adversarial attacks of Deep Learning frameworks are claimed to be due to certain aspects such as the use of single-point estimates. Prior works [4] suggests that traditional neural networks are not well calibrated and the primary reasons for their vulnerability to adversarial attacks are overfitting and making overconfident predictions while allowing the networks to perform well on task in hand.

Attack strategies often evaluate gradients based on input points in order to identify directions of high losses [[3],[7]]. This variability can be intuitively linked to the uncertainty in the prediction, which has drawn attention to Bayesian Neural Networks (BNNs), which some recent studies have argued are a more robust deep learning paradigm [[6], [9], [8], [10]].

In BNNs, the posterior average of the gradients of the loss function vanishes in the large data limit, providing robustness against gradient-based attacks. The magnitude of gradients decreases as more samples are taken from the BNN posterior in various BNN architectures trained with Variational Inference (VI) [11]. The robustness property holds when the ensemble is drawn from the true posterior. However, empirical evidences in [11] shows that it is not likely to be true that the sole ensemble with zero averaging property of gradients is posterior distribution. Bayesian inference methods that provide cheaper approximates such as Variational Inference (VI) may exhibit such properties in practice.

As part of this project, Bayesian inference (i.e., Variational Inference) is incorporated with a few existing Deep Neural Network architectures to create BNN models. The same is checked for robustness against strong state-of-the-art white-box attacks. This project aims to test the listed theoritical hypothesis using six models AlexNet, LeNet, Simple CNN, Bayesian AlexNet, Bayesian LeNet and Bayesian SimpleCNN using 2 datasets - MNIST and CIFAR-10. Then, the models are attacked with five state-of-the-art Gradient-based attacks: $l_\infty - FGSM$, $l_\infty - PGD$, $l_2 - PGD$, BIM attacks and plots for the models' respective test accuracies are presented.

2 Bayesian Neural Network

Bayesian neural networks are popular for their robustness to over-fitting, and their ability to easily learn small datasets. In the form of probability distributions, Bayesian approach further offers uncertainty estimates unlike traditional models. At the same time, this approach integrates it parameters using a prior probability, computing the average across various models during training, which prevents overfitting by providing a regularization effect to the network.

Although theoretically attractive, modelling such a distribution over the filers (kernels) of a Deep Neural Network has been attempted never so successfully before. This is perhaps due to the huge number of parameters in such deep networks.

Further, inferring the posterior distribution of a Bayesian NN even in the case of a small number of parameters is not an easy task. The posteriors of a data distribution can be computed using Bayes' Theorem (Equation 1).

$$P(w|X,Y) = \frac{P(X,Y|w) \cdot P(w)}{P(X,Y)} \tag{1}$$

$$P(X,Y) = \int_{w_0} \dots \int_{w_D} P(X,Y,w)\, dw_0 \dots dw_D \tag{2}$$

However, computing the marginal ($P(X,Y)$) over a dataset, Equation 2 where D denotes the dataset, especially for an image dataset is computationally intractable due to the number of integrals involved. Thus often approximations to the model posterior are used instead, with the variational inference being a popular approach.

2.1 Variational Inference

In Bayesian modeling, we want to be able to sample from the posterior of models given the data. We want to be able to *infer* the latent variables (w) from observed data (X). A function is defined as $y = f(x)$ which gives a predictive output $\{y_1 \dots y_N\}$ for the given inputs $\{x_1 \dots x_N\}$ where N denotes the number of examples. In Bayesian inference, we aim to use a prior distribution over the space $p(f)$ of functions which represents our prior belief of functions expected to have generated the data.

Problem faced in Equation 2 can be alleviated by conditioning the model using a finite set of random variables w. We assume for granted that the model depend on these variables alone and are sufficient to approximate the model. Hence, for a new input example x^*, the predictive distribution can be given by Equation 3.

$$P(y^*|x^*,X,Y) = \int p(y^*|f^*)\, p(f*|x^*,w)\, p(w|X,Y)\, df^*\, dw \tag{3}$$

However, $p(w|X,Y)$ still remains an intractable distribution.

We use a *variational* distribution $q_\phi(w) \approx p(w|X,Y)$ in order to approximate it. $q_\phi(w)$ must be a simple distribution (eg. Gaussian) that can be easily computed and substitute the posterior distribution. To find the optimal $q_\phi(w)$, the distance between the surrogate distribution $q_\phi(w)$ and posterior distribution $p(w|X,Y)$ must be minimized.

We thus minimise the Kullback–Leibler (KL) divergence which is a measure of similarity between two distributions.

$$\begin{aligned} KL(q_\phi(W) \,||\, p(w|X,Y)) &= \int_w q_\phi(w)\, log\left(\frac{q_\phi(w)}{w|X,Y}\right) dw \\ &= \int_w q_\phi(w) \cdot log\left(\frac{q_\phi(w)}{w|X,Y}\right) dw + \int_w q_\phi(w)\, log(p(Y|X,w)) dw \\ &= -\mathbb{E}_{w \sim q_\phi(w)} log\left(\frac{w|X,Y}{q_\phi(w)}\right) + \sum_{(x,y)\in D} \mathbb{E}_{q_\phi(w)} log(p(Y|X,w)) \end{aligned} \tag{4}$$

In the equation above, $KL\ (q(W)\ ||\ p(w|X,Y))$ is a distance metric and hence a positive term. $log(p(X,Y))$ denotes the evidence of the data and is a constant negative term. The expectation term thus becomes a lower bound for the KL divergence and is termed as *Evidence Lower Bound*. Thus, our objective function, ELBO can be maximised w.r.t the variational parameters defining surrogate distribution $q(w)$ in order to minimize KL-divergence.

$$\mathcal{L}(\phi) = -KL(q_\phi(w) \,||\, p(w|X,Y)) + \mathcal{L}_D(\phi) \tag{5}$$

$$where\ \mathcal{L}_D(\phi) = \sum_{(x,y)\in D} \mathbb{E}_{q_\phi(w)} log(p(Y|X,w)) \tag{6}$$

$\mathcal{L}_D\phi$ is the *expected log-likelihood*. Minimizing the negative of Equation 5 is also known as an optimised version of *Bayes by Backprop*

2.2 Local Reparametrisation Trick

There exists various algorithms for the gradient-based optimization of the ELBO (Equation 5) with q and p being differentiable. One of which, *stochastic gradient variational Bayes (SGVB)* method [1], parameterizes the random variable $w \sim q_\phi(w)$ as $w = f(\epsilon, \phi)$ where f is a differentiable function and $\epsilon \sim p(\epsilon)$ is a random noise vector drawn from noise distribution $p(\epsilon)$.The expected log-likelihood can be now be formed by an unbiased differentiable minibatch-based Monte Carlo estimator:

$$\mathcal{L}_D(\phi) \approx \mathcal{L}_D^{(SGVB)}(\phi) = \frac{N}{M}\sum_{1}^{M} logp(y^i|x^i, w^i = f(\epsilon,\phi)) \tag{7}$$

where $(x^i, y^i) \in D$ and M denotes the size of a mini-batch. The above is differentiable w.r.t ϕ and is unbiased. Thus, its gradients is unbiased as well, implying, $\nabla_\phi \mathcal{L}_D(\phi) \approx \nabla_\phi \mathcal{L}_D^{(SGVB)}(\phi)$

Stochastic gradient descent highly depends on the variance of the gradients. For minibatches M, the variances as stated in [1] is given by:

$$Var[\mathcal{L}_D^{(SGVB)}(\phi)] = N^2\left(\frac{1}{M}Var[L_i] + \frac{M-1}{M}Cov[L_i, L_j]\right) \tag{8}$$

where, $L_i = logp(y^i|x^i, w^i = f(\epsilon^i,\phi))$ and $Var[L_i] = Var_{\epsilon,x^i,y^i} logp(y^i|x^i, w^i = f(\epsilon,\phi))$

In [5], an alternative estimator for which $Cov[Li, Lj] = 0$, so that the variance of stochastic gradients scales as 1/M is proposed. *Local Reparameterisation Trick*, global uncertainty in the weights is translated into a form of local uncertainty that is independent across examples by not sampling ϵ directly but only sampling the intermediate variables $f(\epsilon)$ through which ϵ influences $\mathcal{L}_D^{(SGVB)}(\phi)$.

Similar reparameterisation is further applied to sampling weights as well. The weights (also ϵ) influence the expected log likelihood only through the non-linear neuron activations $B = AW$ which are of much lower dimensions. Thus a low-cost Monte Carlo estimator is proposed directly for activations instead of sampling the Gaussian weights and then computing the resulting activations. For a factorized Gaussian posterior on the weights, the posterior for the activations (conditional on the input **A**) is also factorized Gaussian:

$$\begin{gathered} q_\phi(w_{i,j}) = \mathcal{N}(\mu_{i,j}, \sigma^2_{i,j}) \forall w_{i,j} \in \mathbf{W} \implies q_\phi(b_{m,j}|\mathbf{A}) = \mathcal{N}(\gamma_{m,j}, \delta_{m,j}) \\ \gamma_{m,j} = \sum_{neurons} a_{m,i}\mu_{i,j}\ ,\ \delta_{m,j} = \sum_{neurons} a^2_{m,i}\mu^2_{i,j} \end{gathered} \tag{9}$$

Activations can be directly computed using $b_{m,j} = \gamma_{m,j} + \sqrt{\delta_{m,j}}\zeta_{m,j}$ where $\zeta_{m,j} \sim \mathcal{N}(0,1)$. It is shown thatm ζ is a $MXno.\ neurons$ matrix hence we only sample M thousands instead of M millions as in original case.

This results in the subsequent equation for activations b in a convolutional layer:

$$b_j = A_i * \mu_i + \epsilon_j \odot \sqrt{A_i^2 * \mu_i^2} \tag{10}$$

where i, j, w, h represent input, output layers and width and height of the image repectively. A_i represents the receptive field, * represents a convolutional operation and $\odot$ represents element-wise multiplication and $\epsilon \sim \mathcal{N}(0,1)$.

3 Model Configuration

3.1 Frequentist models

We use three CNNs:

1. AlexNet
2. LeNet
3. SimpleCNN (3 convolutional layers and 3 Fully connected layers)

The model architectures are given in Appendix. By default, all the models use ReLU activation function.

3.1.1 Other configurations

We use Adam optimizer for all models and Loss is calculated using Cross Entropy Loss.

3.2 Bayesian models

3.2.1 Activation Function

We apply the Softplus function in Bayesian models, because we want to ensure that the variance term never becomes zero. The Softplus activation function gives a smooth approximation of ReLU. In this application this is has a high analytically important advantage that it never becomes zero for $x \to -\infty$, whereas ReLU becomes zero for any negative input.

$$Softplus(x) = \frac{1}{\beta} \cdot log(1 + exp(\beta \cdot x)) \tag{11}$$

β is set as 1 by default.

3.2.2 Objective Function

As mentioned in Equation 4, The objective function can be summarized as:

$$\mathcal{F}(X, Y, \theta) \approx \sum\ log\ q_\theta(w^i|X, Y)\ -\ log\ p(w^i)\ +\ log\ p(Y|X, w) \tag{12}$$

3.2.3 Variational posterior

$logq_\theta(w^i|X, Y)$ in Equation 4 is defined as the log of variational posterior sampled as a Gaussian distribution:

$$log\ q_\theta(w^i|X, Y) = \sum_i log\ (\mathcal{N}(w_i|\mu, \sigma^2)) \tag{13}$$

where σ is of dimension (input_channel, output_channel, *(kernel_size,kernel_size))

3.2.4 Prior

3.2.5 Variational posterior

$log\ p(w^i)$ in Equation 4 is defined as the log of priors over weight sampled as a product of individual Gaussians:

$$log\ p(w^i) = \sum_i log\ (\mathcal{N}(w_i|0, \sigma^2)) \tag{14}$$

where σ is of dimension (input_channel, output_channel, *(kernel_size,kernel_size))

3.3 Parameter Initialization

A Gaussian distribution is used to store the mean and variance values instead of just one weight. Variance cannot be negative and Softplus as the activation function reinforces it. We express variance σ as σ_i = Softplus(ρ_i) where ρ is an unconstrained parameter. A good method for variance initialization is still unknown. We perform gradient descent over $\theta = (\mu, \rho)$, and individual weight $w_i \sim \mathcal{N}(w_i|\mu_i, \sigma_i)$.

3.4 Adversarial Attacks

In the White-box adversarial attack scenario, the attacker is expected to know the model parameters.The above models are studied under 4 strong white-box attacks:

- l_∞ FGSM
- l_∞ BIM
- l_∞ PGD
- l_2 PGD

- **FGSM**: Fast Gradient Sign Method (FGSM), using loss function, applies perturbations in the direction of the gradient to create an adversarial example x^{adv}. Written as:

$$x^{adv} = x + \epsilon \cdot sign(\nabla_x J(x, y_{true})) \tag{15}$$

  where x is a data point, x^{adv} is the adversarially perturbed image, J is the loss function, y_{target} is the target/true label, and ϵ is the hyperparameter

- **PGD**: Projected Gradient Descent (PGD) initializes uniform random perturbation then runs more iterations until finding an adversarial example. PGD creates a stronger attack than other previous iterative methods.

$$x^{adv}_{i+1} = \prod_{x+S} x^{adv}_t + \alpha \cdot sign(\nabla_x J(x, y_{true})) \tag{16}$$

  where $\prod$ clips the input at positions around $[x^{adv}_t - \epsilon, x^{adv}_t + \epsilon]$ (projection operator), α is the gradient step size and x+S represents the perturbation set.

- **BIM**: FGSM is appliediteratively with small step size, and the pixel values of intermediate results are clipped after each step to ensure that they are in the ϵ-neighbourhood of the original image.

$$\begin{aligned} x^{adv}_0 &= X \\ x^{adv}_{i+1} &= Clip_{X,\epsilon}\{x^{adv}_i + \alpha \cdot sign(\nabla_x J(x, y_{true}))\} \end{aligned} \tag{17}$$

4 Results

In this section, we compare the performance of the baseline models to their Bayesian counterparts. As robustness is generally defined in terms of misclassification for CNNs, we provide results for the same. For small-epsilon attacks we use ϵ values 0 to 0.05 with an increment of 0.005 and for large-epsilon attacks we use ϵ values 0 to 0.5 with an increment of 0.05. ϵ is directly proportional to the perturbations in the adversarial image and hence the strength of the attack. The statistical verification is performed on 10000 images from test dataset of MNIST and CIFAR10 datasets.

Models (AlexNet, Bayesian AlexNet (BAlexNet), LeNet and Bayesian LeNet(BLeNet)) are compared on their performance on MNIST dataset and their respective test accuracies on the true dataset are

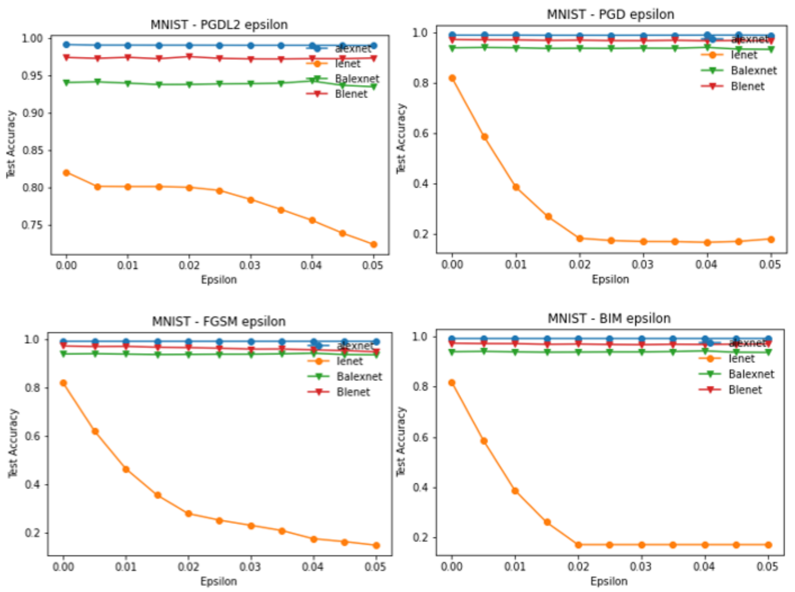

Figure 1: Test Accuracies on adversarial MNIST dataset for small epsilon values

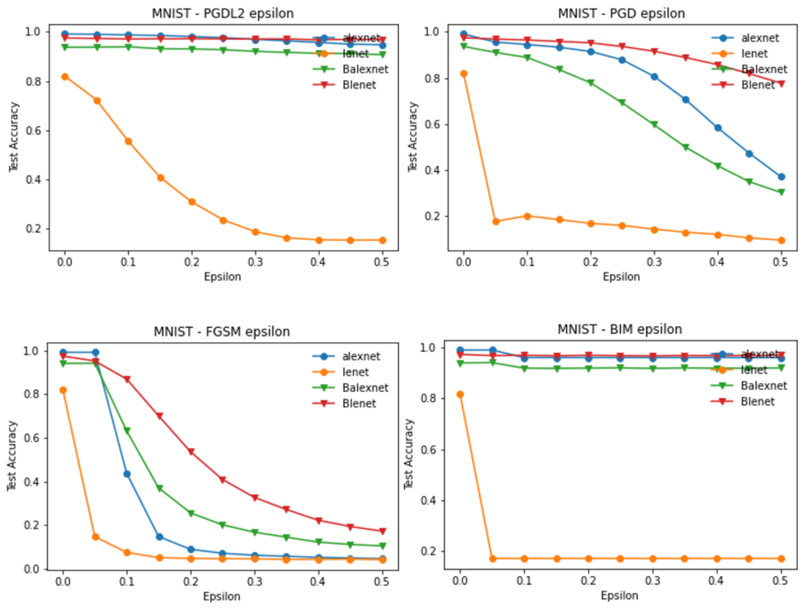

Figure 2: Test Accuracies on adversarial MNIST dataset for large epsilon values

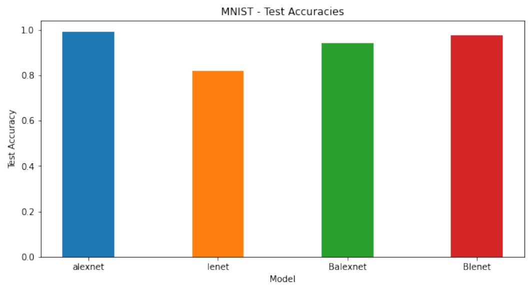

Figure 3: Test Accuracies on original MNIST dataset

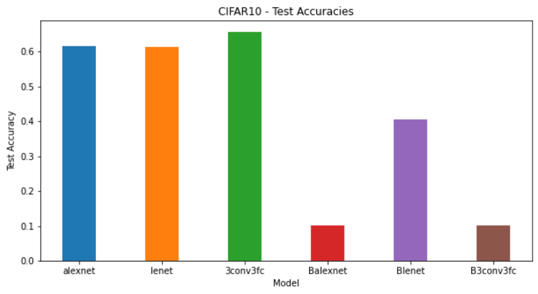

Figure 4: Test Accuracies on original CIFAR10 dataset

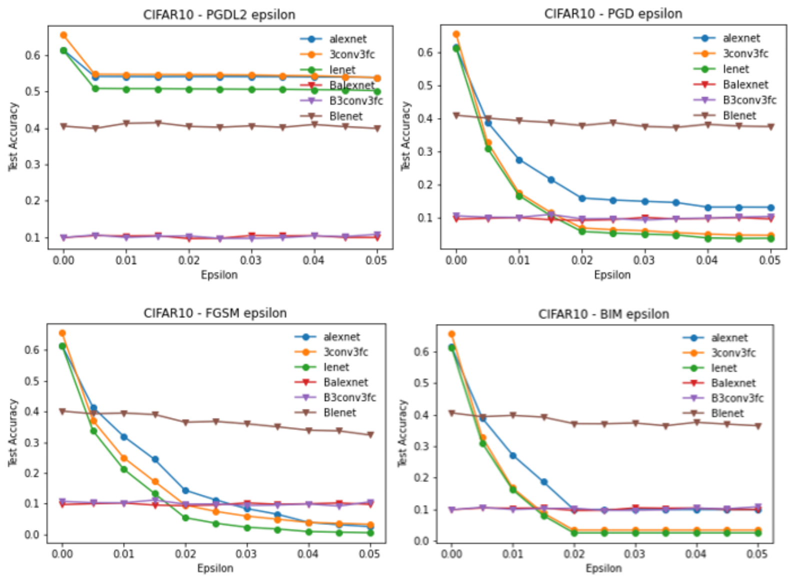

Figure 5: Test Accuracies on adversarial CIFAR10 dataset for small epsilon values

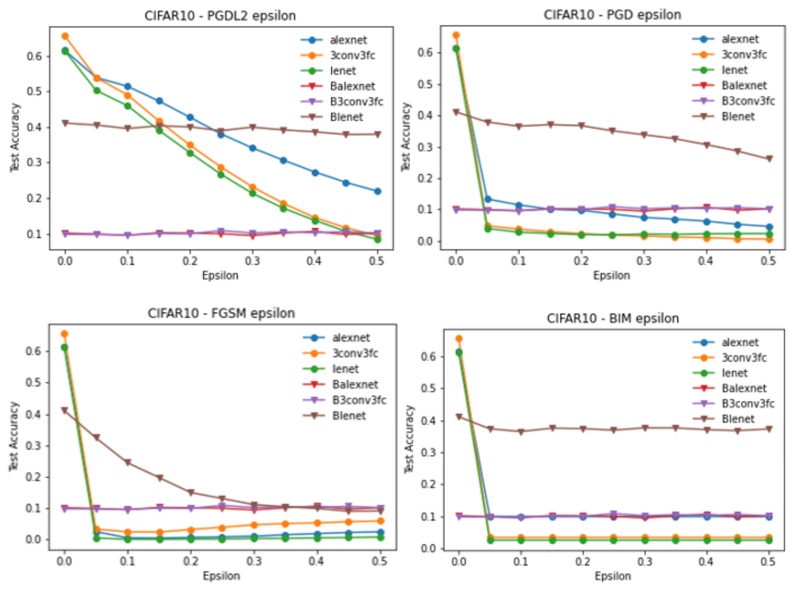

Figure 6: Test Accuracies on adversarial CIFAR10 dataset for large epsilon values

reported in Figure 3. It can be observed that AlexNet and Bayesian LeNet have the best accuracies on original dataset. Further, they are attacked using the mentioned white-box gradient based attacks 3.4 for small and large epsilon values and their test accuracies are compared.

For small perturbation values in Figure 1, frequentist LeNet model is highly vulnerable to all attacks, however, bayesian LeNet is robust to the same, however, AlexNet and the Bayesian models seem to be robust to all the attacks. For large perturbation values in Figure 2, Bayesian models consistently show better robustness on an average to the gradient-based attacks.

Models AlexNet, Bayesian AlexNet (BAlexNet), 3Conc3FC and Bayesian 3conv3fc (Simple CNN), LeNet and Bayesian LeNet(BLeNet) are compared on their performance on CIFAR-10 dataset and their test accuracies on the true dataset are reported in Figure 4. However, two bayesian models, BAlexNet and B3conv3FC fail to perform well on the original data. Further, they are attacked using the mentioned white-box gradient based attacks 3.4 for small and large epsilon values and their test accuracies are compared.

For small perturbation values in Figure 5, despite having a much lower accuracy in original data Bayesian LeNet (BLeNet) model seems to retain its robustness consistently for most attacks. Although, BAlexNet and B3Conv3FC (bayesian Simple CNN) show poor performance, it is notable that the adversarial attacks did not cause their accuracy to go any lower unlike their frequentist counterparts. For large perturbation values in Figure 6, Bayesian LeNet model shows the average best performance consistently displaying its robustness against high perturbations. Frequentist models can be observed to be too vulnerable towards PGD, FGSM and BIM attacks from how their accuracies almost dropped to zero in untargetted attacks.

5 Conclusion

This study showed the probabilistic robustness for BNNs which takes both model and data uncertainty into account, and can be used to capture, among other properties, the robustness of the bayesian models towards gradient-based adversarial examples. Models that are data-driven and robust are an essential component to the construction of effective decision-making AI technologies. In this regard, it is noteworthy that Bayesian ensembles of Neural Networks have the capability to defend against a wide range of gradient-based adversarial attacks.

The results of this study have some significant limitations, yet they are promising. To begin with, Bayesian inference is extremely challenging in large non-linear models. In our hands, cheaper approximations, such as VI, also exhibited some adversarial robustness, though reduced, but there is no guarantee that this will hold in general. Also, this robustness might be more evident in better trained models. Secondly, theoretical evidence of absolute robustness of Bayesian Neural Networks is limited to a thermodynamic large data limit, which has never been realised in practice.

Thirdly, this study focused on four popular gradient-based attack strategies which directly uses gradients in the presented empirical evaluation. Gradient-based attacks which are more complex in nature exist, also non-gradient based and query based attacks. Evaluating towards more wide ranged attacks would confirm Bayesian's robustness.

References

[1] Diederik P Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: (2013). DOI: 10.48550/ARXIV.1312.6114. URL: https://arxiv.org/abs/1312.6114.

[2] Christian Szegedy et al. "Intriguing properties of neural networks". In: *arXiv* (2013). DOI: 10.48550/ARXIV.1312.6199.

[3] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *arXiv* (2014). DOI: 10.48550/ARXIV.1412.6572.

[4] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. "Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images". In: *CoRR* abs/1412.1897 (2014). URL: http://arxiv.org/abs/1412.1897.

[5] Diederik P. Kingma, Tim Salimans, and Max Welling. "Variational Dropout and the Local Reparameterization Trick". In: (2015). DOI: 10.48550/ARXIV.1506.02557. URL: https://arxiv.org/abs/1506.02557.

[6] Reuben Feinman et al. "Detecting Adversarial Samples from Artifacts". In: *arXiv* (2017). DOI: 10.48550/ARXIV.1703.00410.

[7] Aleksander Madry et al. "Towards Deep Learning Models Resistant to Adversarial Attacks". In: *arXiv* (2017). DOI: 10.48550/ARXIV.1412.6572.

[8] Artur Bekasov and Iain Murray. "Bayesian Adversarial Spheres: Bayesian Inference and Adversarial Examples in a Noiseless Setting". In: *arXiv* (2018). DOI: 10.48550/ARXIV.1811.12335.

[9] Yarin Gal and Lewis Smith. "Sufficient Conditions for Idealised Models to Have No Adversarial Examples: a Theoretical and Empirical Study with Bayesian Neural Networks". In: *arXiv* (2018). DOI: 10.48550/ARXIV.1806.00667.

[10] Xuanqing Liu et al. "Adv-BNN: Improved Adversarial Defense through Robust Bayesian Neural Network". In: *arXiv* (2018). DOI: 10.48550/ARXIV.1810.01279.

[11] Ginevra Carbone et al. "Robustness of Bayesian Neural Networks to Gradient-Based Attacks". In: *arXiv* (2020). DOI: 10.48550/ARXIV.2002.04359.

Appendices

Network Architectures

Frequentist models

AlexNet: by default, uses ReLU activation function.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1             [-1, 64, 8, 8]           7,808
              ReLU-2             [-1, 64, 8, 8]               0
           Dropout-3             [-1, 64, 8, 8]               0
         MaxPool2d-4             [-1, 64, 4, 4]               0
            Conv2d-5            [-1, 192, 4, 4]         307,392
              ReLU-6            [-1, 192, 4, 4]               0
         MaxPool2d-7            [-1, 192, 2, 2]               0
            Conv2d-8            [-1, 384, 2, 2]         663,936
              ReLU-9            [-1, 384, 2, 2]               0
          Dropout-10            [-1, 384, 2, 2]               0
           Conv2d-11            [-1, 256, 2, 2]         884,992
             ReLU-12            [-1, 256, 2, 2]               0
           Conv2d-13            [-1, 256, 2, 2]         590,080
             ReLU-14            [-1, 256, 2, 2]               0
          Dropout-15            [-1, 256, 2, 2]               0
        MaxPool2d-16            [-1, 256, 1, 1]               0
           Linear-17                   [-1, 10]           2,570
================================================================
Total params: 2,456,778
Trainable params: 2,456,778
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.31
Params size (MB): 9.37
Estimated Total Size (MB): 9.69
```

LeNet: by default, uses ReLU activation function Figure

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 28, 28]             156
            Conv2d-2           [-1, 16, 10, 10]           2,416
            Linear-3                  [-1, 120]          48,120
            Linear-4                   [-1, 84]          10,164
            Linear-5                   [-1, 10]             850
================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.05
Params size (MB): 0.24
Estimated Total Size (MB): 0.29
----------------------------------------------------------------
```

SimpleCNN: by default, uses Softplus activation function 3.2.1

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]             832
          Softplus-2           [-1, 32, 32, 32]               0
         MaxPool2d-3           [-1, 32, 15, 15]               0
            Conv2d-4           [-1, 64, 15, 15]          51,264
          Softplus-5           [-1, 64, 15, 15]               0
        MaxPool2d-6             [-1, 64, 7, 7]                0
           Conv2d-7            [-1, 128, 5, 5]          204,928
         Softplus-8            [-1, 128, 5, 5]                0
        MaxPool2d-9            [-1, 128, 2, 2]                0
     FlattenLayer-10                  [-1, 512]               0
           Linear-11                 [-1, 1000]         513,000
         Softplus-12                 [-1, 1000]               0
           Linear-13                 [-1, 1000]       1,001,000
         Softplus-14                 [-1, 1000]               0
           Linear-15                   [-1, 10]          10,010
================================================================
Total params: 1,781,034
Trainable params: 1,781,034
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 1.68
Params size (MB): 6.79
Estimated Total Size (MB): 8.48
----------------------------------------------------------------
```

Bayesian models

BAlexNet: by default, uses Softplus activation function 3.2.1

```
BBBAlexNet(
  (conv1): BBBConv2d()
  (act1): Softplus(beta=1, threshold=20)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): BBBConv2d()
  (act2): Softplus(beta=1, threshold=20)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): BBBConv2d()
  (act3): Softplus(beta=1, threshold=20)
  (conv4): BBBConv2d()
  (act4): Softplus(beta=1, threshold=20)
  (conv5): BBBConv2d()
  (act5): Softplus(beta=1, threshold=20)
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (flatten): FlattenLayer()
  (classifier): BBBLinear()
)
```

```
----------------------------------------------------
        Layer (type)               Output Shape
====================================================
         BBBConv2d-1             [-1, 64, 8, 8]
          Softplus-2             [-1, 64, 8, 8]
         MaxPool2d-3             [-1, 64, 4, 4]
         BBBConv2d-4            [-1, 192, 4, 4]
          Softplus-5            [-1, 192, 4, 4]
         MaxPool2d-6            [-1, 192, 2, 2]
         BBBConv2d-7            [-1, 384, 2, 2]
          Softplus-8            [-1, 384, 2, 2]
         BBBConv2d-9            [-1, 256, 2, 2]
         Softplus-10              [-1, 256, 2, 2]
        BBBConv2d-11              [-1, 128, 2, 2]
         Softplus-12              [-1, 128, 2, 2]
        MaxPool2d-13              [-1, 128, 1, 1]
     FlattenLayer-14                    [-1, 128]
        BBBLinear-15                     [-1, 10]
----------------------------------------------------------------
```

#MNIST

BLeNet: by default, uses Softplus activation function 3.2.1

```
BBBLeNet(
  (conv1): BBBConv2d()
  (act1): Softplus(beta=1, threshold=20)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): BBBConv2d()
  (act2): Softplus(beta=1, threshold=20)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (flatten): FlattenLayer()
  (fc1): BBBLinear()
  (act3): Softplus(beta=1, threshold=20)
  (fc2): BBBLinear()
  (act4): Softplus(beta=1, threshold=20)
  (fc3): BBBLinear()
)
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape
================================================================
         BBBConv2d-1            [-1, 6, 28, 28]
          Softplus-2            [-1, 6, 28, 28]
         MaxPool2d-3            [-1, 6, 14, 14]
         BBBConv2d-4           [-1, 16, 10, 10]
          Softplus-5           [-1, 16, 10, 10]
         MaxPool2d-6             [-1, 16, 5, 5]
      FlattenLayer-7                  [-1, 400]
         BBBLinear-8                  [-1, 120]
          Softplus-9                  [-1, 120]
        BBBLinear-10                   [-1, 84]
         Softplus-11                   [-1, 84]
        BBBLinear-12                   [-1, 10]
----------------------------------------------------------------
```

BSimpleCNN: by default, uses Softplus activation function 3.2.1

```
BBB3Conv3FC(
  (conv1): BBBConv2d()
  (act1): Softplus(beta=1, threshold=20)
  (pool1): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): BBBConv2d()
  (act2): Softplus(beta=1, threshold=20)
  (pool2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): BBBConv2d()
  (act3): Softplus(beta=1, threshold=20)
  (pool3): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (flatten): FlattenLayer()
  (fc1): BBBLinear()
  (act4): Softplus(beta=1, threshold=20)
  (fc2): BBBLinear()
  (act5): Softplus(beta=1, threshold=20)
  (fc3): BBBLinear()
)

----------------------------------------------------------------
```

```
         Layer (type)               Output Shape
========================================================
          BBBConv2d-1          [-1, 32, 32, 32]
           Softplus-2          [-1, 32, 32, 32]
          MaxPool2d-3          [-1, 32, 15, 15]
          BBBConv2d-4          [-1, 64, 15, 15]
           Softplus-5          [-1, 64, 15, 15]
          MaxPool2d-6            [-1, 64, 7, 7]
          BBBConv2d-7           [-1, 128, 5, 5]
           Softplus-8           [-1, 128, 5, 5]
          MaxPool2d-9           [-1, 128, 2, 2]
      FlattenLayer-10                 [-1, 512]
        BBBLinear-11                [-1, 1000]
         Softplus-12                [-1, 1000]
        BBBLinear-13                [-1, 1000]
         Softplus-14                [-1, 1000]
        BBBLinear-15                  [-1, 10]
--------------------------------------------------------
```