[Re] Latent Weights Do Not Exist: Rethinking Binarized Neural Network Optimization

Nik Vaessen
vaessen@kth.se

Peter Mastnak
mastnak@kth.se

Sri Datta Budaraju
budaraju@kth.se

Abstract

This report is on the reproduciblity of "Latent Weights Do Not Exist: Rethinking Binarized Neural Network Optimization" published at NeurIPS 2019. The paper proposes a novel view on how Binary Neural Networks are trained and propose a new algorithm called *Bop* (Binary Optimization) implemented in TensorFlow. In this report, we reproduce the experiment which explores the effect of the hyperparameters τ and γ used in *Bop* by reimplementing it in PyTorch. We also reproduce training a deep binary neural network on CIFAR-10 and achieve a similar accuracy. The repository of our code is hosted at `https://github.com/bsridatta/Rethinking-Binarized-Neural-Network-Optimization`.

1 Introduction

Contemporary Deep Neural Networks are trained and deployed on GPU-accelerated computing instances hosted in data centers. While achieving groundbreaking results in fields such as Computer Vision[8], Speech Recognition[1], NLP[6][12] and Reinforcement Learning [10], they require expensive GPU hardware while also consuming a large amount of energy [3]. By restricting weights and activations to $\{-1, 1\}$, Binary Neural Networks (BNNs) have a substantially lower energy requirement at run-time compared to equivalent architectures with real-valued weights and activation [5]. This allows them to be deployed in situations where computational power, energy and network access to data centers are limited. Applications such as self-driving vehicles, autonomous drones, mobile devices in areas with underdeveloped infrastructure or robotic space exploration meet those constraints and could benefit from deploying BNNs.

Currently, BNNs are trained with "latent weights". A real-valued network with parameters $\theta_r \in \mathbb{R}$ is initialized. During the forward pass the binary weights $\theta_b = sign(\theta_r)$ are used to compute a prediction. In the backward pass, the gradient $\frac{\delta \mathcal{L}}{\delta \theta_r}$ of the loss function $\mathcal{L}$ with respect to real-valued weights θ_r is computed. Consequently θ_r can be updated with an optimization strategy such as SGD or Adam. After convergence, the binary weights $\theta_b = sign(\theta_r)$ represent the trained BNN. See section 2.1 for a detailed description.

[9] proposes a novel view on "latent weights" and presents a state-of-the-art method for BNN optimization called *Binary optimization (Bop).* They argue that a "latent weight" $w_r \in \theta_r$ can be better understood as

$$w_r = |w_r| * sign(w_r) := magnitude * \{-1, 1\}$$

that is, it defines both a magnitude and a sign. The sign represents the binary weight w_b in the network, while the magnitude encodes some inertia against flipping the sign of w_b during the training process. Based on this view, *Bop* trains a BNN by keeping track of the moving average m with adaptivity rate γ of the gradient $\frac{\delta \mathcal{L}}{\delta w_b}$. When m exceeds a threshold τ, the sign of w_b is flipped. This process is simpler than the real-valued counterpart in that involves only 2 hyperparameters to tune (γ

and τ). It also requires less memory during training compared to optimizing a real-valued weight with SGD and momentum, or Adam. See section 2.2 for a detailed description.

The aim of this article is to report on the reproducibility of [9], hereafter referred to as *the paper* or [9]. To reproduce the experiments of the paper the *Bop* algorithm was re-implemented in the PyTorch framework from scratch. We reproduced section 5.1 (the effect of the hyperparameters τ and γ) and section 5.2 (training on CIFAR-10) of the paper, while ignoring section 5.3 (training on Imagenet). We also conducted some additional experiments not outlined in the original paper which assisted the authors of [9] to make some theoretical claims.

This reproducibility report is structured as follows. Section 2 contains a detailed description of "latent-weight" and *Bop* optimization. Section 3 describes the experiments which were conducted to verify reproducibility and section 4 reports on the results of those experiments. Lastly, in section 5 the results are analyzed and section 6 concludes the reproducibility report.

2 Methodology

2.1 Binary Neural Networks

In this section, we go into the details of how the literature describes the binarization of neural networks. The principle idea is to convert the real-valued weights, as well as the activations, to either -1 or +1. To binarize mentioned real-valued weights both [5] and [4] use two different binarization functions. First, they use a deterministic binarization function:

$$x_{bin} = sign(x) = \begin{cases} +1 & if\, x \geq 0 \\ -1 & otherwise \end{cases} \tag{1}$$

Where x_{bin} is the binary value and x is the real value. The second binarization function is stochastic:

$$x_{bin} = \begin{cases} +1 & with\, probability\, p = \sigma(x) \\ -1 & with\, probability\, 1-p \end{cases} \tag{2}$$

Where σ is the "hard sigmoid" function:

$$\sigma(x) = clip\left(\frac{x+1}{2}, 0, 1\right) = \max\left(0, \min\left(1, \frac{x+1}{2}\right)\right) \tag{3}$$

In [5] they achieved excellent results using hard sigmoid function as it is not as computationally expensive as the soft version. Currently in practice, deterministic binarization is used more often. To use stochastic binarization, random bits generator needs to be used for quantization which is hard to implement [4].

In both cases, if we use stochastic or deterministic binarization, we will encounter the same problem when propagating gradients after discretization. The derivative of the sign function is going to be zero. To overcome this problem Hinton in his 2012 lecture 15b proposed a "Straight-Through Estimator" [11]. Courbariaux et al. in [4] use a similar implementation of a straight-through estimator as Hinton except that they take into account the saturation effect and they use a deterministic rather than a stochastic sampling of the bit. Given that we got the estimator g_q of the gradient $\frac{\partial C}{\partial q}$ the straight-through estimator of the gradient $\frac{\partial C}{\partial r}$ is then:

$$g_r = g_q 1_{|r| \leq 1} \tag{4}$$

From equation 4 we can see that the gradient is canceled when r gets bigger or equal to 1, that is done in order to prevent the performance drop [4].

We need to compute the gradients only during the training phase, these latent weights are not used in run-time and thus do not defeat the purpose of binarizing the network. The behaviour of the network does not change as long as the sign of the latent weight does not change. These weights can be viewed as an inertia for the network and as this inertia increases stronger gradient signal is required to flip the binary weight and thus contributing to the stability of the network. From this point of view, the optimizer is rather altering the inertia of the network rather than the binary weights.

2.2 Binary Optimization

The latent weight free binary optimizer referred to as *Bop* is specifically designed for binary neural networks. The optimizer is mainly based on 3 intuitions.

First, as the only goal of the latent weights and the optimizer is to decide whether or not to flip a particular weight, the desired optimizer should be able to clearly address this task. Secondly, gradient signal consistency is an important feature. It could be inferred by taking the previous gradients into consideration by averaging the gradient signals over training steps. Third, strength is equally important as consistency. The strength is the magnitude of the gradient signal and ignoring the weak signal reduces the noise in the training phase. The consistent signals are obtained by exponential moving average of the gradient over training steps by:

$$m_t = (1-\gamma)m_{t-1} + \gamma g_t = \gamma \sum_{r=0}^{t} (1-\gamma)^{t-r} g_r \tag{5}$$

where, m_t is the moving average at time t, γ is the adaptivity rate and g is the gradient at time t. Higher the adaptivity rate, the easier it is to adapt to the new gradient, it could be related to learning rate. We flip the weight if this exponential moving average, the consistency, is more than a certain threshold τ.

$$w_t^i = \begin{cases} -w_{t-1}^i & if \ \ \left|m_t^i\right| \geq \tau \ \ and \ \ sign\left(m_t^i\right) = sign\left(w_{t-1}^i\right) \\ w_{t-1}^i & otherwise. \end{cases} \tag{6}$$

where w_{t-1} is weight before computing the exponential moving average from the gradient. Thus we use γ to control the consistency of the gradient to be considered and τ to control the strength of the gradient that would affect the behaviour of the binary network.

2.3 Reproducing experiments of [9] with their code

As a sanity check, we ran the experiments of [9] using their code, which uses TensorFlow and Larq[1]. We ran the experiments on Google Cloud Platform on Google's Deep Learning Virtual Machine with Cuda 10.0 and 1 NVIDIA K100 GPU. As the Deep Learning Virtual Machine is based on Debian 9, only python 3.5 was available. To run the author's code requires python $>=$ 3.6, we manually compiled python 3.6.4 from source to run the code. We added a PR [2] to [9]'s code repository to aid other people who are interested in running the code on GCP. We got the equivalent results as published in [9].

2.4 Re-implementation in PyTorch

The authors only implement the binary optimizer from scratch. The binary convolutional and fully connected layers were taken from Larq. As this library only supports TensorFlow we also had to implement the following from scratch: the binary optimizer, binary fully connected layer, binary 2d convolutional layer, a custom sign function which computes a clipped straight-through estimator as gradient and the metric for keeping track of the number of flips each epoch. Our implementation uses PyTorch Lightning [7] to bootstrap the implementation of the training and testing pipeline.

3 Experiments

3.1 Theoretical claims

The authors of [9] mention two experiments which aided their understanding of "latent weight" optimization. In the first experiment, they challenge the viewpoint hypothesis that a binary weight vector acts as an approximation to the real-valued weight vector. The second experiment shows

[1]See `https://larq.dev`. This framework exposes an API for building BNN's with Keras. It's built and released by the authors of [9].

[2]See `https://github.com/plumerai/rethinking-bnn-optimization/pull/17`.

empirical proof for a corollary that states that under "latent weight" optimization, the learning rate can be set arbitrary for each weight as long as its initialization is scaled accordingly. However, a detailed description of these experiments is not included in the paper and they are also not included in the accompanying source code. The following subsections detail the steps we took as the best guess for reproducing the experiments.

3.1.1 Approximation viewpoint

To challenge the approximation viewpoint hypotheses, [9] only mentions that they trained a BNN with latent-weight optimization. Following that, they evaluate the trained network with both the real-valued "latent weights" and the binary weights while using the sign function as the activation function. They also mention that they retrain the batch-normalization layers on the real-valued "latent weight" network. They observed that the real-valued latent weights do not perform better than the binary weights, even after retraining the batch statistics.

This experiment was replicated on the MNIST dataset retrieved via the Tensorflow Keras dataset API[3] All pixel values were normalized between -1 and 1. The network was built with the Larq and Tensorflow Keras framework [2]. The network architecture was 3 binary 2d-convolution blocks with 32, 64 and 64 filters, 3x3 kernel and stride 1. Each convolution block also included a 2x2 max pooling and a batch normalization layer. The convolution blocks were followed by 2 binary linear blocks with 64 and 10 nodes. Each binary linear block also included a batch normalization layer. All binary layers had their "latent weights" clipped to $[-1, 1]$. The activation function was the sign function (Equation 1), which was applied after batch normalization. The last linear layer had a softmax activation instead. Adam was used for optimization with default *learning rate* $= 0.001$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The loss function was sparse categorical cross entropy. Training was done on a CPU. No train/validation split was applied. The weights of the last epoch were used to calculate test accuracy.

First, a model with binary weights is trained from scratch with latent-weight optimization. The accuracy is then evaluated on both the binary weights and the "latent" real-valued weights. Then the binary layers are frozen, and the network is retrained so that only the batch normalization layers are updated. The expectation is that both the real-valued evaluation and the real-valued evaluation with retrained batch statistics perform equal or worse in comparison with the binary weight evaluation.

3.1.2 Arbitrary learning rate

Theorem 1 in paper [9] states that the binary weights of BNNs trained with "latent" weights are invariant to the learning rate of the optimizer as long as the initial "latent" weights are scaled accordingly. It is also required that the pseudo gradient Φ mentioned in equation 4 does not depend on the magnitude of the weights.

To show that the advantage of several learning rates can also be achieved by scaling the initial "latent" real-valued weights, the following experiment was conducted. A BNN is trained on MNIST with the same architecture and training process as described in section 3.1.1. However, different configurations of learning rate, weight initialization and optimisation strategy were tested:

1. Glorot initialization with SGD and learning rate 0.01
2. Glorot Uniform initialization with SGD and learning rate 1
3. Glorot Uniform initialization scaled by 0.01 with SGD and a learning rate 0.01
4. Uniform random initialization with SGD and learning rate 0.01
5. Uniform random initialization with Adam and learning rate 0.01, $\beta_1 = 0.9$ and $\beta_2 = 0.999$

The expectation is that configuration 2 and 3 perform equally well due to the scaled initialization, while configuration 1 performs differently from 2 and 3. We expect that configuration 1 and 4 perform equally well, as the magnitude of the weights (which differs between initialization strategies) has no influence on the pseudo gradient.

[3]See `https://www.tensorflow.org/api_docs/python/tf/keras/datasets`.

3.2 Effect of hyperparameters

Binary optimization, as described in section 2.2, requires two hyperparameters: The threshold τ and the moving average adaptivity rate γ. To explore the behaviour of these hyperparameters, a network was trained with 6 different combinations: A constant $\tau = 10^{-6}$ with varying $\gamma = [10^{-2}, 10^{-3}, 10^{-4}]$ and a constant $\gamma = 10^{-3}$ with varying $\tau = [0, 10^{-5}, 10^{-6}]$. The network and training procedure were identical to what is described in section 3.3 except for the number of epochs. During training the number of flips in the sign of the weights per optimization step was monitored. Because this value is very large at the start of training but tends to zero, the value is reported in the natural log domain according to

$$\pi_t = ln(\frac{\textit{sign changes at step t}}{\textit{total number of parameters}} + 10^{-9}) \tag{7}$$

The term 10^{-9} prevents a natural log of 0 when no weights flipped.

The experiment was conducted on Google Cloud Platform with using Google's Deep Learning VM and a single NVIDIA V100 GPU with CUDA 10.1, and PyTorch 1.3.0 using our PyTorch re-implementation.

3.3 Training on CIFAR-10

The authors of [9] trained a VGG-inspired binary neural network on CIFAR-10. The exact architecture was unclear and was retrieved from their source implementation. The network architecture is as follows:

$(\ \ 3 \times 128\ BC3) - BatchNorm - (128 \times 128\ BC3) - BatchNorm - MaxPool$
$(128 \times 256\ BC3) - BatchNorm - (256 \times 256\ BC3) - BatchNorm - MaxPool$
$(256 \times 512\ BC3) - BatchNorm - (512 \times 512\ BC3) - BatchNorm - MaxPool$
$1024\ BFC - BatchNorm - 1024\ BFC - BatchNorm - 10\ BFC - BatchNorm - Softmax$

$x \times y$ BC3 is a binary convolutional layer with a x input channels, y output channels, 3x3 kernel and stride 1. x BL is a binary fully connected layer with x units. Activation is accomplished by the fact that all convolutional and fully connected layers apply the sign function (equation 1) on their input except for the first layer, which accepts the real-valued normalized input data.

The data was received from the torchvision datasets API[4]. The last 10% of the training data was used for validation. The training data was preprocessed by applying a 4x4 padding, taking a random 32x32 crop, applying a random horizontal flip with probability 0.50 and normalizing between $[-1, 1]$. The test data was only normalized between $[-1, 1]$. Binary weights were initialized by first creating real-valued weights with He-initialization[5] and then applying the sign function. *Bop* was used as optimization method for the binary layers with threshold $\tau = 10^{-8}$ and adaptivity rate $\gamma = 10^{-4}$ for 500 epochs. The adaptivity rate was decayed with 0.1 every 100 epochs. The training was halted early when no weight flips had occurred in 10 consecutive epochs. The batch normalization layers were optimized with Adam with default parameters *learning rate* $= 0.001$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The weights with the lowest validation loss were used to compute the test accuracy.

The experiment was conducted on Google Cloud Platform with using Google's Deep Learning VM and a single NVIDIA V100 GPU with CUDA 10.1, and PyTorch 1.3.0 using our PyTorch re-implementation.

[4]See `https://pytorch.org/docs/stable/torchvision/datasets.html`.
[5]This is the default method for linear and convolutional layers in PyTorch

4 Results

4.1 Theoretical claims

4.1.1 Approximation viewpoint

First, the binary weights are trained from scratch for 6 epochs. The accuracy is then evaluated on both the binary weights and the "latent" real-valued weights. Then the binary layers are frozen, so that only the batch normalization layers are retrained for an additional 6 epochs.

The result of evaluating on the test set of MNIST can be seen in Table 1. The binary weights show an accuracy of 0.97, which is decreased to 0.10 when using the "latent" real-valued weights instead. However, retraining the batch statistics makes the "latent" real-weights outperform the binary weights with an accuracy of 0.98.

Table 1: Training a Binary CNN on MNIST with 10 random initializations. The *binary model* is evaluated with binarized weights, while the *real model* is evaluated without binarizing the weights. The *real retrained model* is trained with all but the batch normalization layers frozen.

Model	Mean	Variance
Binary	0.9663	3.2e-4
Real	0.1039	3.4e-6
Real Retrained	0.9807	1.81e-6

4.1.2 Arbitrary learning rate

The trends in training the BNN with different configurations for 100 epochs can be seen in Figure 1. We observe that uniform random, Adam and $lr = 0.01$ performs best. It's closely matched in performance by Glorot, SGD, $lr = 1$ and Glorot scaled by 0.01, SGD, $lr = 0.01$. Note that the learning rate of 1 is equal to a learning rate of 0.01 with weights rescaled by 0.01. Note also that Glorot, SGD and $lr = 0.01$ without the re-scaled initialization performs much worse. There is no difference between using Glorot initialization or uniform random initialization with SGD and learning rate 0.01.

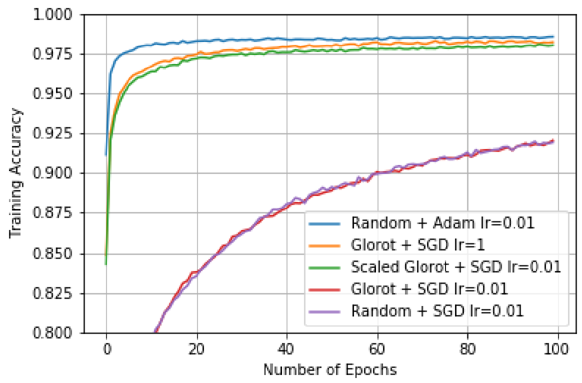

Figure 1: Training BNN on MNIST for 100 epochs with latent weights optimization for different configurations of optimization strategies and weight initializations

4.2 Effect of hyperparameters

Figure 2 shows the result of exploring the behaviour of γ and τ. This experiment matches the hyperparameter analysis from section 5.1 in the paper [9] and achieved similar results. We check the accuracies of different values of γ and τ in *Bop* for BinaryNet on CIFAR-10 (train: $-$, validation: $--$). The training curve is very noisy to illustrate and compare with validation curve and is thus smoothened with 1D Gaussian filter with a standard deviation of 100. Figure 2 shows that a very high γ or a very low τ results in high number of flips and thus a very noisy training deviating the model from convergence. Whereas a very low γ and high τ results in a very slow and stable learning with close to zero weight flips.

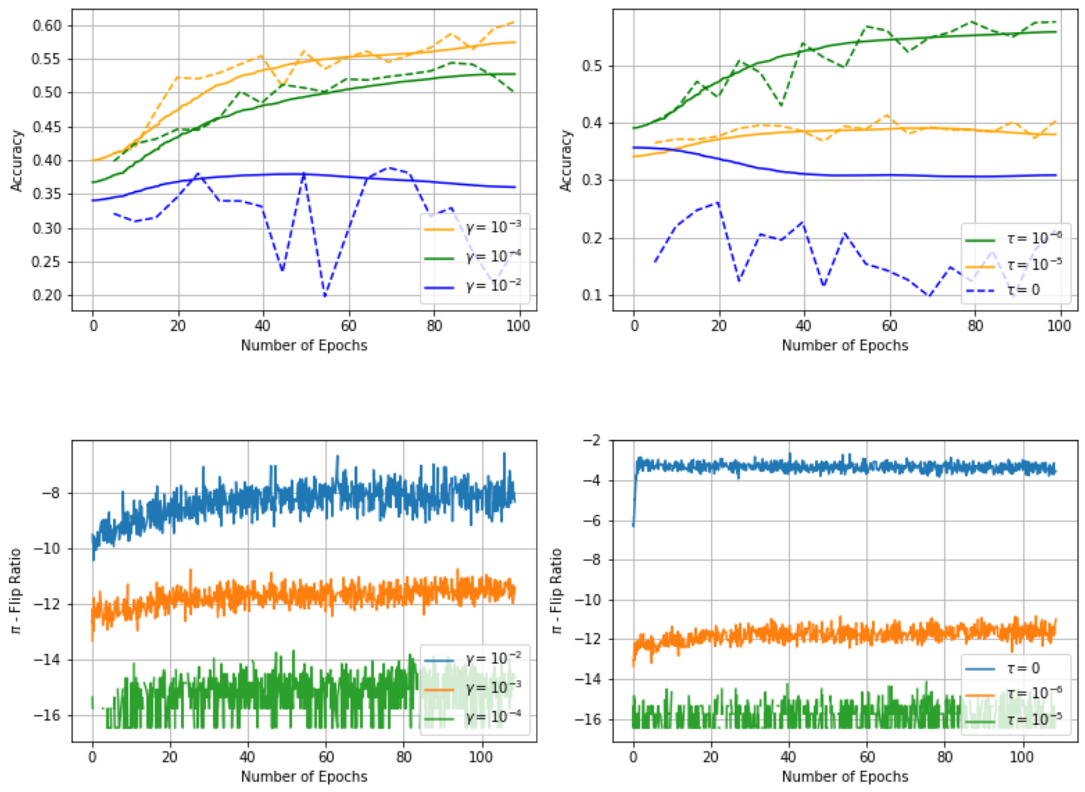

Figure 2: On the left γ is fixed at 10^{-3} and we compare three values τ. On the right τ is fixed at 10^{-6} and we compare three values for γ. The tensorboard graphs can be found at `https://tensorboard.dev/experiment/up4UbhojT6uZKjnKPyZaRQ`

4.3 Training on CIFAR-10

The training and validation accuracy of training on CIFAR-10 for 327 epochs can be seen in Figure 3. We can observe that the flip ratio of the entire network falls significantly as the adaptivity rate γ decreases. After 300 epochs when γ is 10^{-6} no weights are flipped with two exceptions where only 1 weight is flipped across all the layers. Therefore training was stopped after 327 epochs instead of running for 500 epochs. The training and validation accuracy trends similar to the figures from [9]. Our network achieves a test accuracy of **88.67**% with the best model on the validation set. This is lower than the accuracy of **91.3**% presented in [9].

5 Discussion

We were confronted with several issues during re-implementing the code based on the description of *Bop* in the paper. Without batch normalization, binary neural networks can barely outperform random predictions. When debugging *Bop* we thought that we made a mistake in implementing the algorithm, while we erred by testing on a straight forward 3-layer fully connected network without batch normalization layers. The second issue we ran into when implementing *Bop* was that batch normalization layers are (by definition) non-binary as the mean and variance must be real-valued.

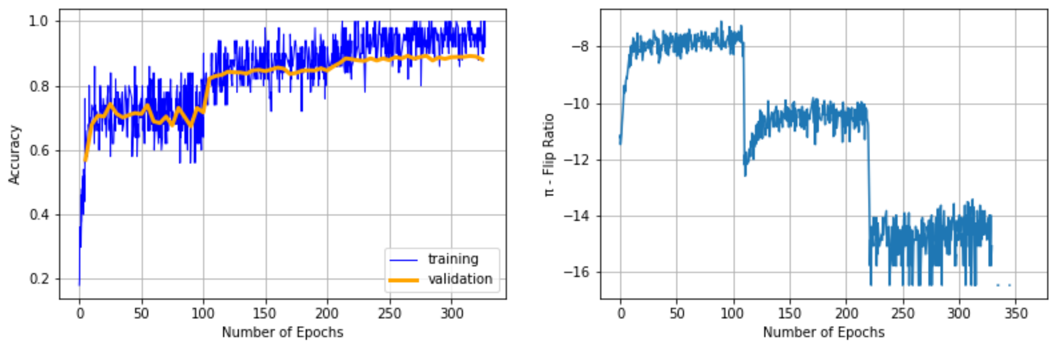

Figure 3: On the left are the training and validation accuracies and on the right are the flip ratios of the overall network while training on CIFAR-10. The tensorboard graphs can be found at `https://tensorboard.dev/experiment/OXOeA57LTVWBtYzGRSdwVA`.

Thus an implementation of *Bop* must still use a method like SGD or Adam to update the batch normalization parameters and needs to be intertwined with the binary optimization strategy.

Our experiment to disprove the approximation viewpoint was inconclusive. While simply using the real-valued weights instead of the binary weights lead to an accuracy of 0.10 (random predictions), retraining the batch statistics improved the accuracy by almost 1.5% compared to the binary weights. Perhaps this experiment should be conducted on more difficult data sets or different network architectures. Our experiment does not disprove the approximation viewpoint, but indicates that batch normalization does almost all of the work in solving MNIST with BNN's.

From our experiments to verify the arbitrary learning rate, we observe that effect of scaling the learning rate can be reversed by also scaling the initial weights. We also see that a BNN trained using SGD with learning rate of 0.01 has almost the same trends when trained with Glorot and uniform random initialization. This supports the theorem 1 in the paper which states that the pseudo gradient does not depend on the magnitude of the weights. This is due to the fact that the initialization techniques mostly contribute towards the magnitudes of the weights and not the signs. This experiment also supports the statement made in [9] that Adam is preferred over SGD for training a BNN with latent weights.

We didn't reach the same accuracy with the optimal hyperparameters on CIFAR-10 stated in [9]. Our implementation makes use validation data, while their implementation most likely trains on the full training dataset. Their maximum test accuracy might also be retrieved from an exhaustive search over all weights, as they plot the test accuracy for every epoch. It would be valuable to get insights into the difficulty of finding correct hyperparameters. We observe the training of BNNs with *Bop* to be extremely sensitive to the hyperparameters γ and τ. A lot of combinations we tried while debugging lead to a single step flipping a high percentage of weights and all consecutive steps flipping 0 weights. This does make it easy to stop a particular hyperparameter combination early.

6 Conclusion

We show that [9] is reproducible to a large extent. The architecture details and implementation of binary layers, both of which are not related to *Bop*, are not mentioned in the paper and had to be found by performing a literature study on Binary Neural Networks. Due to the fact that the authors of [9] published the source code of their experiments, there was an additional way to gain this information. We were unable to reproduce the proof that the approximation viewpoint is wrong. We will need to contact the authors for more information regarding their experiments, which is currently not possible yet on OpenReview.

Acknowledgments

The authors would like to thank CodeOcean for providing computational resources to the authors under to the NIPS 2019 reproducibility challenge. They would also like to thank Google for providing a Google Cloud Platform educational grant to the KTH course *DD2412 Deep Learning, Advanced Course* under which the authors participated in the NIPS 2019 reproducibility challenge.

References

[1] Dario Amodei et al. “Deep speech 2: End-to-end speech recognition in english and mandarin”. In: *International conference on machine learning*. 2016, pp. 173–182.

[2] François Chollet et al. *Keras*. `https://keras.io`. 2015.

[3] Adam Coates et al. “Deep learning with COTS HPC systems”. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 1337–1345. URL: `http://proceedings.mlr.press/v28/coates13.html`.

[4] Matthieu Courbariaux and Yoshua Bengio. “BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1”. In: *CoRR* abs/1602.02830 (2016). arXiv: `1602.02830`. URL: `http://arxiv.org/abs/1602.02830`.

[5] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In: *CoRR* abs/1511.00363 (2015). arXiv: `1511.00363`. URL: `http://arxiv.org/abs/1511.00363`.

[6] Jacob Devlin et al. “Fast and Robust Neural Network Joint Models for Statistical Machine Translation”. In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Baltimore, Maryland: Association for Computational Linguistics, June 2014, pp. 1370–1380. DOI: `10.3115/v1/P14-1129`. URL: `https://www.aclweb.org/anthology/P14-1129`.

[7] W.A. Falcon. *PyTorch Lightning*. `https://github.com/williamFalcon/pytorch-lightning`. 2019.

[8] Yuying Ge et al. *DeepFashion2: A Versatile Benchmark for Detection, Pose Estimation, Segmentation and Re-Identification of Clothing Images*. 2019. arXiv: `1901.07973 [cs.CV]`.

[9] Koen Helwegen et al. “Latent Weights Do Not Exist: Rethinking Binarized Neural Network Optimization”. In: *arXiv preprint arXiv:1906.02107* (2019).

[10] Matteo Hessel et al. *Rainbow: Combining Improvements in Deep Reinforcement Learning*. 2017. arXiv: `1710.02298 [cs.AI]`.

[11] Geoffrey Hinton. *Neural networks for machine learning. In: Coursera Video Lectures (2012).*

[12] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to Sequence Learning with Neural Networks”. In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 3104–3112. URL: `http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf`.