UNDERSTANDING WEIGHT-MAGNITUDE HYPERPARAMETERS IN TRAINING BINARY NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Binary Neural Networks (BNNs) are compact and efficient by using binary weights instead of real-valued weights. Current BNNs use latent real-valued weights during training, where several training hyper-parameters are inherited from real-valued networks. The interpretation of several of these hyperparameters is based on the magnitude of the real-valued weights. For BNNs, however, the magnitude of binary weights is not meaningful, and thus it is unclear what these hyperparameters actually do. One example is weight-decay, which aims to keep the magnitude of real-valued weights small. Other examples are latent weight initialization, the learning rate, and learning rate decay, which influence the magnitude of the real-valued weights. The magnitude is interpretable for real-valued weights, but loses its meaning for binary weights. In this paper we offer a new interpretation of these magnitude-based hyperparameters based on higher-order gradient filtering during network optimization. Our analysis makes it possible to understand how magnitude-based hyperparameters influence the training of binary networks which allows for new optimization filters specifically designed for binary neural networks that are independent of their real-valued interpretation. Moreover, our improved understanding reduces the number of hyperparameters, which in turn eases the hyperparameter tuning effort which may lead to better hyperparameter values for improved accuracy.

1 INTRODUCTION

A learnable weight of a Binary Neural Network (BNN) is a single bit: -1 or $+1$. These binary weights are compact to store and allow efficient execution. With suitable hardware, BNNs can be deployed on tiny devices with limited computational capacity, enabling important applications on, for example, edge devices.

Training BNNs using gradient decent is difficult because of the discrete binary values. Thus, BNNs are often optimized with so called 'latent', real-valued weights (De Putter & Corporaal, 2022), which allow continuous optimization. These latent real-valued weights can then be discretized to -1 or $+1$ by, *e.g.*, taking the positive or negative sign of the real value. Using real-valued latent weights is the current practice in training BNNs (Kim et al., 2021b; Liu et al., 2020; Martinez et al., 2020).

Hyperparameters are important for training BNNs with latent real-valued weights. The latent weights link to several essential hyperparameters, such as their initialization, learning rate, learning rate decay, and weight decay.

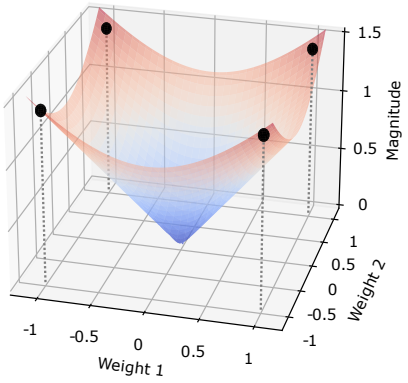


Figure 1: The magnitude for real-valued and binary weights. Changes in the real-valued weights change their magnitude. For binary weights, however, the magnitude will never change. Thus, magnitude-based hyperparameters need reinterpretation for binary weights.

These hyperparameters are important for BNNs, as shown for example in the strong baseline of Martinez et al. (2020), which improves BNN accuracy by better tuning these hyperparameters. Another example is the binary ReActNet of Liu et al. (2020), where better tuning the latent weight hyperparameters improves results as reported by Liu et al. (2021a). Tuning the hyperparameters for the latent weights is essential for good accuracy in BNNs.

In this paper we investigate the latent weight hyperparameters used in a BNN: initialization, learning rate, learning rate decay, and weight decay. All these hyperparameters influence the magnitude of the latent weights. Yet, in a BNN, the binary weights are -1 or $+1$, and as illustrated in Figure 1, always have a constant magnitude and the magnitude-based hyperparameters lose their meaning. We draw inspiration from the seminal work of Helwegen et al. (2019), who reinterpret latent weights from an inertia perspective and state that latent weights do not exist. Thus, the magnitude of latent weights also does not exist. Here, we investigate what latent weight-magnitude hyperparameters mean for a BNN, how they relate to each other, and what justification they have. We make the following contributions: 1. A gradient filtering perspective on latent weight hyperparameters; 2. Through the filtering perspective offer a clear understanding of magnitude-based hyperparameters; 3. A justification of which magnitude-based hyperparameters to use; 4. Offer a simplified setting, with fewer hyperparameters to tune, achieving similar accuracy as current, more complex methods.

2 Related Work

Latent weights in BNNs. By tying each binary weight to a latent real-valued weight, continuous optimization approaches can be used to optimize binary weights. Some methods minimize the quantization error between a latent weight and its binary variant (Rastegari et al., 2016; Bulat & Tzimiropoulos, 2019). Others focus on gradient approximation (Liu et al., 2018; Lee et al., 2021; Zhang et al., 2022), or on reviving dead weights (Xu et al., 2021; Liu et al., 2021b) or a loss-aware binarization (Hou et al., 2017; Kim et al., 2021a). These works directly apply traditional optimization techniques inspired by real-valued network such as weight decay, learning rate and its decay, and optimizers. The summary of De Putter & Corporaal (2022) gives a good overview of these training techniques in BNNs. Recently, some papers (Liu et al., 2021a; Martinez et al., 2020; Hu et al., 2022) noticed that the interpretation of these optimization techniques does not align with the binary weights of a BNNs (Lin et al., 2017; 2020). Here, we aim to shed light on why, by explicitly analyzing latent weight-magnitude hyperparameters in a BNN.

Optimization by gradient filtering. Gradient filtering is a common approach used to tackle the noisy gradient updates caused by minibatch sampling. Seminal algorithms including Momentum (Sutskever et al., 2013) and Adam (Kingma & Ba, 2015) which use a first order infinite impulse response filter (IIR), *i.e.* exponential moving average (EMA) to smooth noisy gradients. In binary network optimization, Bop (Helwegen et al., 2019) and its extension (Suarez-Ramirez et al., 2021) introduce a threshold to compare with the smoothed gradient by EMA to determine whether to flip a binary weight. In our paper, we build on second order gradient filtering techniques to reinterpret the hyperparameters that influence the latent weight updates.

Latent weight magnitudes. Several techniques exploit the magnitude of the latent weights during BNN optimization. Latent weights clipping is proposed in (Courbariaux et al., 2015) and followed by its extensions (Alizadeh et al., 2018; Hubara et al., 2016) to clip the latent weights within a $[-1, 1]$ interval to prevent the magnitude of latent weights from growing too large. Gradient clipping (Cai et al., 2017; Courbariaux et al., 2015; Qin et al., 2020) stops gradient flow if the magnitude of latent weight is too large. Work on latent weight scaling (Chen et al., 2021; Qin et al., 2020) standardizes the latent weights to a pre-defined magnitude. Here, we analyze the role of the latent weight magnitudes on BNN optimization.

Two step training. Excellent results are achieved by a two-step training strategy (Liu et al., 2021a; 2020) that in the first step trains the network from scratch using only binarizing activations with weight decay, and then in the second step they fine-tune by training without weight decay. Our method reinterprets the meaning of the magnitude based weight decay hyperparameter in optimizing BNNs from a filtering perspective, and we can achieve on similar performance as two step training with a simpler setting, using just a single step.

3 A GRADIENT FILTERING ANALYSIS OF LATENT WEIGHT-MAGNITUDE HYPERPARAMETERS

In this section we will show our derivation of our filtering based BNN optimizer. We start the analysis in a BNN with latent weights, as they have shown great results in practice, but convert it to an equivalent latent-weight free setting, as in Helwegen et al. (2019). To do this we need to start in a magnitude invariant setting, which means that no gradient-clipping, latent-weight clipping or scaling based on the channel-wise mean of the latent-weights is used.

BNN setup. We use Stochastic Gradient Descent (SGD) with weight decay and momentum as a starting point, as this is a commonly used setting, see Rastegari et al. (2016), Liu et al. (2018), Qin et al. (2020). Our setup is as follows:

$$w_0 = \text{init}(), \tag{1}$$
$$m_i = (1-\gamma)m_{i-1} + \gamma \nabla_{\theta_i}, \tag{2}$$
$$w_i = w_{i-1} - \beta_i \epsilon (m_i + \lambda w_{i-1}), \tag{3}$$
$$\theta_i = \text{sign}(w_i), \tag{4}$$
$$\text{sign}(x) = \begin{cases} -1, & \text{if } x < 0; \\ +1, & \text{if } x > 0; \\ \text{random}\{-1,+1\} & \text{otherwise.} \end{cases} \tag{5}$$

Here, w_i is a latent weight at iteration i which is initialized at w_0. θ_i is a binary weight, ϵ is the learning rate, β_i is the learning rate decay factor from a decay scheduler, λ is the weight decay factor, γ is the momentum exponentially moving average discount factor, ∇_{θ_i} is the gradient over the binary weight and random$\{-1,+1\}$ is a uniformly randomly sampled -1 or +1.

We then convert to the latent-weight free setting of Helwegen et al. (2019) where latent weights are interpreted as accumulating negative gradients. We introduce $g_i = -w_i$, which allows working with gradients instead of with latent weights. We can then write Eq 3 as follows

$$g_i = g_{i-1} + \beta_i \epsilon (m_i - \lambda g_{i-1}). \tag{6}$$

Latent weight initialization. To investigate latent weight initialization we unroll the the recursion in Eq 6 by writing it out as a summation:

$$g_i = (1 - \beta_i \epsilon \lambda) g_{i-1} + \beta_i \epsilon m_i, \tag{7}$$
$$= g_0 + \epsilon \sum_{r=0}^{i} (1 - \beta_r \epsilon \lambda)^{i-r} \beta_r m_i. \tag{8}$$

The term g_0 in Eq 8, takes the role of the latent weight initialization. Since there no longer is a latent-weight, it no longer makes sense to use real-valued weight initialization techniques (Glorot & Bengio, 2010; He et al., 2015). Instead, g_0 can be initialized according to other gradient filtering techniques such as Momentum (Sutskever et al., 2013) and Adam (Kingma & Ba, 2015). We follow these methods, and simply initialize $g_0 = 0$. To prevent all binary weights to start at the same value, we use the stochastic sign function in Eq 5 that randomly chooses a sign when the input is exactly 0.

Learning rate and weight decay. The learning rate ϵ appears in two places in Eq 8: once to the left outside of the summation, and once inside the summation.

With $g_0 = 0$, the leftmost ϵ can only scale the latent weight and will not influence outcome of the sign in Eq 4 as

$$\text{sign}\left(\epsilon \sum_{r=0}^{i} (1 - \beta_r \epsilon \lambda)^{i-r} \beta_r m_i\right) = \text{sign}\left(\sum_{r=0}^{i} (1 - \beta_r \epsilon \lambda)^{i-r} \beta_r m_i\right). \tag{9}$$

Thus, when using a 0 initialization, the leftmost ϵ can be removed, or set to any arbitrary value without influencing the training process.

For the ϵ inside the summation of Eq 8, it appears together with the weight decay term λ. Thus, there are two free hyperparameters that only control one factor, therefore one of them is redundant and can use a single combined hyperparameter $\alpha = \epsilon\lambda$. Instead of setting a value for the learning rate ϵ, and setting a value for the weight decay λ, we now only have to set a single value for α.

Learning rate decay. To see the role of the learning rate decay β_i, we use Eq 9 to freely scale the sum with any value, and we can scale it with α, as

$$g_i = \alpha \sum_{r=0}^{i} (1 - \beta_r \alpha)^{i-r} \beta_r m_i, \tag{10}$$

which allows us to write it as an exponential moving average (EMA) as

$$g_i = (1 - \beta_i \alpha) g_{i-1} + \beta_i \alpha m_i. \tag{11}$$

This shows that for BNNs under magnitude invariant conditions, SGD with weight decay is little more than a simple exponential moving average. This gives a magnitude-free justification for using weight decay since its actual role is to act as the discount factor in an EMA. Note that it is not longer possible to set α to 0 since then there are no updates anymore, but setting to a small (10^{-20}) number will essentially work the same. Now, the learning rate decay directly scales the α, so from now on we will refer to it as α decay. The meaning of α is now clear, as in the EMA it controls how much to take the previous output into account relative to the new input. This essentially determines how far back in time past gradients are taken into account which is similar to the window size of a sliding window average.

Momentum. Now adding back the momentum term of Eq 2 in the original setup yields

$$m_i = (1 - \gamma) m_{i-1} + \gamma \nabla_{\theta_i}, \tag{12}$$

$$g_i = (1 - \beta_i \alpha) g_{i-1} + \beta_i \alpha m_i, \tag{13}$$

$$\theta_i = -\text{sign}(g_i). \tag{14}$$

We see here that SGD with weight decay and momentum is equivalent to smoothing the gradient twice with an EMA filter.

Latent weight optimization as a second order linear infinite impulse response filter. EMAs are a specific type of linear Infinite Impulse Response (IIR) Filter (Proakis, 2001). Linear filters are filters that compute an output based on a linear combination of current and past inputs and past outputs. The general definition is given as a difference equation:

$$y_t = \frac{1}{a_0}(b_0 x_t + b_1 x_{t-1} + ... + b_P x_{t-P} - a_1 y_{t-1} - a_2 y_{t-2} - ... - a_P y_{t-Q}), \tag{15}$$

where t is the time step, y_t are the outputs, x_t are the inputs, a_i and b_i are the filter coefficients and P and Q are the maximum of iterations the filter looks back at the inputs and outputs to compute the current output. The maximum of P and Q defines the order of the filter. An EMA only looks at the previous output and the current input, so is therefore a first order IIR filter. Expressing an EMA as a filter looks as follows:

$$y_t = (1 - \alpha) y_{t-1} + \alpha x_t = \frac{1}{a_0}(b_0 x_t - b_1 \cdot x_{t-1} - a_1 y_{t-1}), \quad b = \begin{bmatrix} \alpha \\ 0 \end{bmatrix}, a = \begin{bmatrix} 1 \\ \alpha - 1 \end{bmatrix}. \tag{16}$$

In our optimizer we have a cascade of two EMAs applied in series to the same signal. Two cascaded linear filters can also be represented by a filter with the order being the sum of the orders of the original filters. To get the new a and b vectors the original ones are convolved with each other. In our case this gives:

$$b = \begin{bmatrix} \gamma \\ 0 \end{bmatrix} \star \begin{bmatrix} \alpha \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha\gamma \\ 0 \\ 0 \end{bmatrix}, \quad a = \begin{bmatrix} 1 \\ \gamma - 1 \end{bmatrix} \star \begin{bmatrix} 1 \\ \alpha - 1 \end{bmatrix} = \begin{bmatrix} 1 \\ (\alpha - 1) + (\gamma - 1) \\ (\alpha - 1) \cdot (\gamma - 1) \end{bmatrix}, \tag{17}$$

when applied to our gradient filtering setting in Eq 13 gives the difference equation:

$$g_i = \beta_i \alpha \gamma \nabla_{\theta_i} - (\beta_i \alpha + \gamma - 2) g_{i-1} - (\beta_i \alpha - 1)(\gamma - 1) g_{i-2}. \tag{18}$$

This shows that in a magnitude invariant setting, SGD with weight decay and momentum is equivalent to a 2nd order linear IIR filter. You can also see that α and γ have the same function. When not using α decay, it would be possible to swap the values for α and γ, without effecting anything. This filtering perspective opens up new methods of analysis for optimizers.

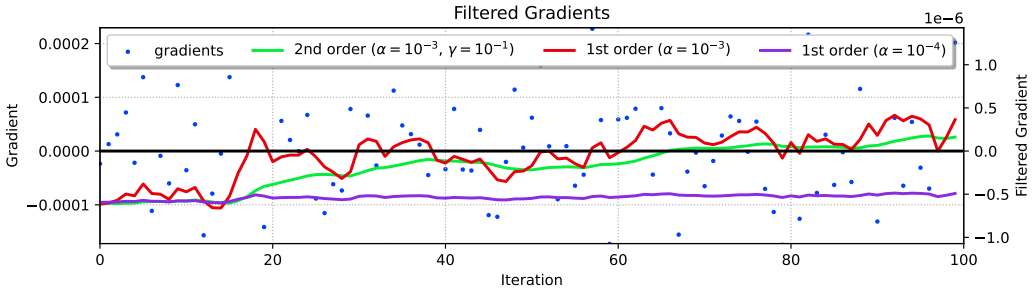
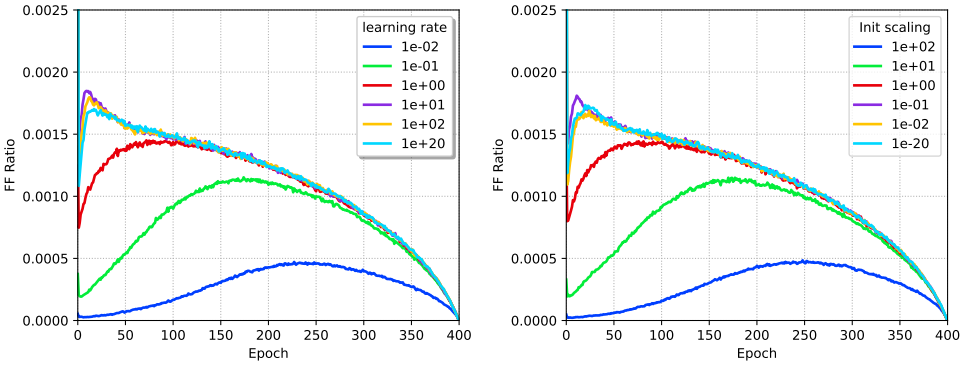


Figure 2: Gradients and filtered gradients using a first and second order filter for a single epoch of training on CIFAR-10. For better visualisation, the filter outputs are scaled up to a similar range as the unfiltered gradients. It can be seen that the unfiltered gradients are noisy and that the filtered outputs are smoother. The second order filter reduces the noise even further compared to the first order filter.



(a) Varying learning rates, constant init.

(b) Vary init, constant learning rate (ϵ=1.0).

Figure 3: In the magnitude independent setting, scaling the learning rate has the exact same effect on the flipping ratio as scaling the initial latent-weights by the inverse.

4 Experiments

We empirically validate our analysis on CIFAR-10, using the BiRealNet-20 architecture. Unless mentioned otherwise the networks were optimized using SGD for both the real-valued and binary parameters with as hyperparameters: learning rate=0.1, momentum with $\gamma = (1 - 0.9)$, weight decay=10^{-4}, batch size=256 and cosine learning rate decay and cosine alpha decay. In our analysis we use the weight flip ratio at every update or the FF ratio (Liu et al. (2021a)).

$$\mathbf{I}_{\text{FF}} = \frac{|\text{sign}(w_{t+1}) - \text{sign}(w_t)|_1}{2} \qquad \mathbf{FF}_{\text{ratio}} = \frac{\sum_{l=1}^{L} \sum_{w \in W_l} \mathbf{I}_{\text{FF}}}{N_{total}} \tag{19}$$

where w_t is a latent weight at time t, L the number of layers, W_l the weights in layer l, and N the total number of weights.

1st order vs 2nd order We visually compare filter orders by sampling real gradients from a single binary weight trained on CIFAR-10 in Figure 2. For the same α, a 1st order filter is more noisy than a 2nd order filter. This may cause the binary weight to oscillate, even though the larger trend is that it should just flip once. To reduce these oscillations with a 1st order filter requires a smaller alpha. This, however, causes other problems because α determines the window size of past gradients and with a smaller α many more gradients are used. This means that it takes much longer for a trend in the gradients to effect the binary weight. Instead, the 2nd order filter has for both benefits: it can filter out high frequency noise while reacting quicker to changing trends.

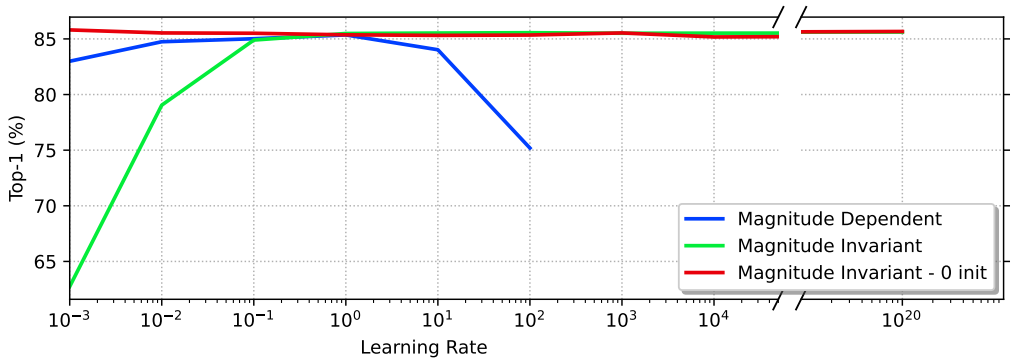


Figure 4: Learning rate effect on accuracy for three settings. (a). standard SGD. (b) Magnitude invariant in Eq 3, by removing clipping and scaling. (c) Magnitude invariant with latent weight initializes of 0 in Eq 8. Setting (a) has just a single optimum. The accuracy in setting (b) is sensitive to small learning rates. For setting (c), accuracy is independent of the learning rate.

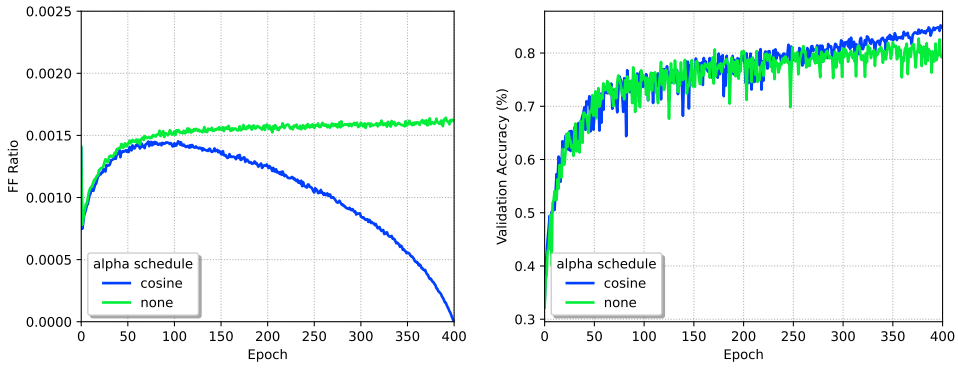


Figure 5: Flipping rate and accuracy for comparing alpha decay vs no alpha decay. When using cosine alpha decay the FF ratio goes towards zero over time while without decay, the flipping will continue, causing the network to not fully converge leading to reduced accuracy.

Learning rate vs initialization In Figure 3 we show the BNN bit flipping ratio with respect to the learning rate and the initialization. Multiplying the learning rate with a certain factor a has the same effect as multiplying the initial weights with the inverse of a: $\text{sign}\left(w_0 - a\epsilon\sum_{r=0}^{i}(1-\beta_r\epsilon\lambda)^{i-r}\beta_r m_i\right) = \text{sign}\left(\frac{1}{a}w_0 - \sum_{r=0}^{i}(1-\beta_r\epsilon\lambda)^{i-r}\beta_r m_i\right)$, because the magnitude of a term inside a sign function does not matter. The larger learning rates in Figure 3(a) and the smaller initialization values in Figure 3(b) are invariant to scaling and have similar flipping ratios. The smaller learning rates and larger initializations do not reach the same flipping ratios, because the initialization vs learning rate ratio is insufficient to update the binary weights. For sufficiently large initialization vs learning rate ratios it means that scaling both the learning rate and the initial latent-weights has no effect on training, but also that scaling the learning rate or scaling the initial latent-weights with the inverse is identical: as seen by comparing the two plots in Figure 3.

Evaluating learning rate and magnitude invariance with 0 initialization We evaluate SGD in the standard magnitude dependent setting with clipping and scaling vs a magnitude independent setting with initializing the latent-weights to zero. To keep the effect of weight decay constant, we scale the weight decay factor inversely with the learning rate. The results in Figure 4 show that when using the standard setting the learning rate has to be carefully balanced. When the learning rate is too small the updates are too small relative to the initial weights, and the network does not properly learn. When the learning rate is too large, then the latent-weights will hit the clipping region

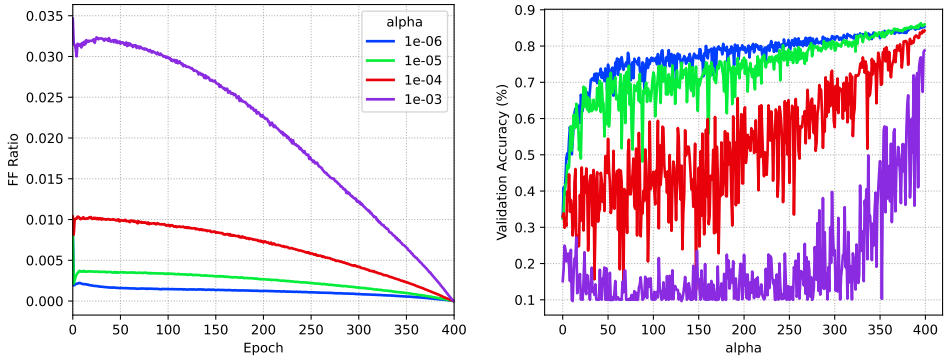


Figure 6: FF ratio and accuracy for varying alphas. BNNs are sensitive to alpha, similar to how they are sensitive to weight decay, because it has a big influence on the FF ratio, which really affects performance. A too high FF ratio leads to noisy training and poor results in the first half of training, while not having enough time to converge to a good result in the second half. On the other hand, a too low FF ratio leads to quick convergence to a good result in the first half of training, but then improves little in the second half.

and will stop updating. When switching to the magnitude invariant setting, however, the clipping problem is solved, because there no longer is a clipping function applied on the gradients. However, the problem of a too small learning rate is enhanced, because the magnitudes of the gradients are smaller when not using the scaling factor and the accuracy drops significantly for small learning rates. When initializing to zero this problem disappears, because there is no initial weight to hinder training and all learning rates perform equally.

Alpha To show the role of α from equation 18 we trained multiple networks with only different weight decay factors. The results in Figure 6 show that alpha has a lot of influence on the training process. Too big values lead to too many binary weight flips every update, which hinders the network from learning. To small and the network converges to quickly to a specific local minima, which hurts the end result. The optimum is choosing something in between that allows for steady improvement during training and seems to be a trade off between exploration and exploitation of the search space.

Alpha decay Keeping the concept of learning rate decay and transforming it to alpha decay is important. For proper convergence the FF ratio should go down towards zero. To achieve this, the alpha decays over time to force the network to converge. This happens because β_i from equation 18 starts at 1 at the start of training and over the epochs decreases towards 0. The specific way this happens is determined by the alpha decay schedule. Figure 5 shows this in practice, one network has been trained with cosine alpha decay and one without any alpha decay. With and without alpha decay both seem to perform well at the start of training, however, the variant without alpha decay plateaus at the end of training while the BNN with alpha decay converges better and continues improving, leading to a better end result.

Equivalent interpretation Here we empirically validate that the SGD setting using latent weights in Eq 3 is equivalent to our gradient filtering interpretation in Eq 18 that no longer uses latent weights. We compare both settings with matching hyperparameters in Figure 7 which shows that these settings are empirically equivalent.

4.1 Comparison with SOTA methods

CIFAR-10: We train all networks for 400 epochs. As data augmentation we use padding of 4 pixels, followed by a 32x32 crop and random horizontal flip. All experiments are performed with Bi-RealNet-20 with the architecture meant for the CIFAR datasets as described in He et al. (2016). For the real-valued parameters and latent-weights when used, we use SGD with a learning rate of 0.1 with cosine decay, momentum of 0.9 and on the non-BN parameters a weight decay of 10^{-4}. When using our optimizer we used an alpha of 10^{-3} with cosine decay and a gamma of 10^{-1}. The

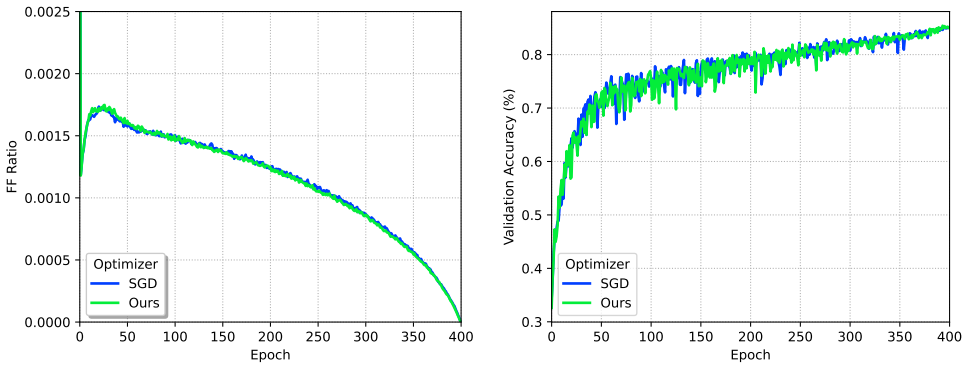


Figure 7: Flipping rate and accuracy for SGD with latent weights in Eq 3 and our gradient-filtering optimizer without latent weights in Eq 18. The methods are empirically equivalent.

results can be seen in Table 1. We outperform SGD with standard settings on the same architecture, but only match performance of IR-Net. We also trained Bi-RealNet-20 with the BABW two step training strategy from Liu et al. (2021a) and see that we get close in perfomance with only one step using our optimizer.

Table 1: Comparison with state-of-the-art on CIFAR-10. The $\star$ denotes that we re-ran these experiments ourselves.

Method	Training Strategy	Bit-width (W/A)	Top-1 Acc(%)
FP		32/32	91.7
DoReFa-Net (Zhou et al., 2016)	One step	1/1	79.3
DSQ (Gong et al., 2019)		1/1	84.1
IR-Net (Qin et al., 2020)		1/1	**86.5**
Bi-Real* (Liu et al., 2018)		1/1	85.0
Bi-Real + Our optimizer		1/1	**86.5**
Bi-Real* (Liu et al., 2018)	Two step	1/1	**86.7**

Imagenet: We base our settings for training on Liu et al. (2021a). We train for 600K iterations with a batch size of 510. For the real-valued parameters we use Adam with a learning rate of 0.0025 with linear learning rate decay. For the binary parameters we use our 2nd order filtering optimizer with $\alpha = 10^{-5}$, which we decay linearly and $\gamma = 10^{-1}$. Since we no longer use latent-weights, we also do not use two-step training to pre-train the latent-weights.

As can be seen in Table 2, ReActNet-A with our optimizer improves upon the state-of-the-art for one step training. It also performs comparable to the state-of-the-art for two step training, by improving 0.3% over ReActNet-A, but still falling 0.8% short of AdamBNN, while only needing one step for training and using slightly less OPs because no channel-wise scaling factor is used.

5 Discussion and Limitations

One limitation of our work is that we do not achieve "superior performance" in terms of accuracy. Our approach merely matches the state of the art results. Note, however, that our goal is to provide insight into how SGD and its hyperparameters behave. We also ended up with an optimizer with less hyperparameters, that also have a better explanation in the context of BNN optimization leading to simpler, more elegant methods.

Table 2: Comparison with state-of-the-art on Imagenet

Method	Training Strategy	BOPs ($\times 10^9$)	FLOPs ($\times 10^8$)	OPs ($\times 10^8$)	Top-1 Acc(%)	Top-5 Acc(%)
CI-BCNN (Wang et al., 2019)	One step	–	–	1.63	59.9	84.2
Binary MobileNet (Phan et al., 2020b)		–	–	1.54	60.9	82.6
MoBiNet (Phan et al., 2020a)		–	–	0.52	54.4	77.5
EL (Hu et al., 2022)		–	–	–	56.4	–
MeliusNet29 (Bethge et al., 2020)		5.47	1.29	2.14	65.8	–
ReActNet-A + Our optimizer		4.82	0.07	0.82	**69.7**	**88.9**
StrongBaseline (Martinez et al., 2020)	Two step	1.68	1.54	1.63	60.9	83.0
Real-to-Binary (Martinez et al., 2020)		1.68	1.56	1.83	65.4	86.2
ReActNet-A (Liu et al., 2020)		4.82	0.12	0.87	69.4	88.6
ReActNet-A-AdamBNN (Liu et al., 2021a)		4.82	0.12	0.87	**70.5**	**89.1**

Another perceived limitation is that our new proposed optimizer can be projected back to a specific setting within the current SGD with latent-weights interpretation. Thus, our analysis might not be needed. While it is true that latent-weights can also be use, we argue that there is no disadvantage to switching to the filtering perspective, because the options are the same, but the benefit is that the hyperparameters make more sense. The option to project back to latent-weights also works the other way around and for those who already have a well tuned SGD optimizer could use it to make it easier to switch to our filtering perspective.

Its also true that our method cannot use common techniques based on the magnitude such as weight clipping or gradient clipping. Yet, we do not really think these techniques are necessary. We see such methods as heuristics to reduce the bit flipping ratio over time, which helps with convergence. However, in our setting, this can also be done using a good α decay schedule without reverting to such heuristics, making the optimization less complex.

We did not yet have the opportunity to test the filtering-based optimizer on more architectures, with only having tested Bi-RealNet on CIFAR-10 and ReActnet-A on Imagenet. However, since our optimizer is equivalent to a specific setting of SGD, we would argue that architectures that have been trained with SGD will probably also work well with our optimizer. This is also a reason why we chose to use ReActNet-A, since it was trained using Adam in both in the original paper (Liu et al., 2020) and in Liu et al. (2021a). The latter specifically argues that Adam works better for optimizing BNNs, but we suspect that the advantages of Adam are diminished because the adaptive learning rate might work the same way in the magnitude invariant setting. However, more research would be needed to be sure.

5.1 Ethics Statement

We believe that this research does not bring up major new potential ethical concerns besides any concerns that might already exist for BNNs. Our work makes training BNNs easier, which might increase their use in practice.

5.2 Reproducibility Statement

We will release all our code to simplifying the reproducibility of our experiments. Two important things for better reproducing our results rely on the GPUs and the dataloader. The reproduction of our ImageNet experiments is not trivial. First, as the teacher-student model is used in our ImageNet experiments, it will occupy much GPU memory. We trained on 3 NVIDIA A40 GPUs, each A40 has 48 GB of GPU memory, with a batch size of 170 per GPU for as much as ten days. Second, for faster training on ImageNet, we used NVIDIA DALI dataloader to fetch the data into GPUs for the image pre-processing. This dataloader could effect training as it uses a slightly different image resizing algorithm than the standard PyTorch dataloader. To keep results consistent with other methods, we do the inference with the standard PyTorch dataloader.

REFERENCES

Milad Alizadeh, Javier Fernández-Marqués, Nicholas D Lane, and Yarin Gal. An empirical study of binary neural networks' optimisation. In *International conference on learning representations*, 2018.

Joseph Bethge, Christian Bartz, Haojin Yang, Ying Chen, and Christoph Meinel. Meliusnet: Can binary neural networks achieve mobilenet-level accuracy? *arXiv preprint arXiv:2001.05936*, 2020.

Adrian Bulat and Georgios Tzimiropoulos. Xnor-net++: Improved binary neural networks, 2019.

Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5918–5926, 2017.

Tianlong Chen, Zhenyu Zhang, Xu Ouyang, Zechun Liu, Zhiqiang Shen, and Zhangyang Wang. " bnn-bn=?": Training binary neural networks without batch normalization. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 4619–4629, 2021.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NeurIPS*, 2015.

Floran De Putter and Henk Corporaal. How to train accurate bnns for embedded systems? *arXiv preprint arXiv:2206.12322*, 2022.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, pp. 249–256, 2010.

Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4852–4861, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pp. 630–645. Springer, 2016.

Koen Helwegen, James Widdicombe, Lukas Geiger, Zechun Liu, Kwang-Ting Cheng, and Roeland Nusselder. Latent weights do not exist: Rethinking binarized neural network optimization. *Advances in neural information processing systems*, 32, 2019.

Lu Hou, Quanming Yao, and James T Kwok. Loss-aware binarization of deep networks. *ICLR*, 2017.

Jie Hu, Ziheng Wu, Vince Tan, Zhilin Lu, Mengze Zeng, and Enhua Wu. Elastic-link for binarized neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 942–950, 2022.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.

Dohyung Kim, Junghyup Lee, and Bumsub Ham. Distance-aware quantization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 5271–5280, 2021a.

Hyungjun Kim, Jihoon Park, Changhun Lee, and Jae-Joon Kim. Improving accuracy of binary neural networks using unbalanced activation distribution. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 7862–7871, 2021b.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015.

Junghyup Lee, Dohyung Kim, and Bumsub Ham. Network quantization with element-wise gradient scaling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6448–6457, 2021.

Mingbao Lin, Rongrong Ji, Zihan Xu, Baochang Zhang, Yan Wang, Yongjian Wu, Feiyue Huang, and Chia-Wen Lin. Rotated binary neural network. *ECCV*, 2020.

Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *NeurIPS*, 2017.

Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *ECCV*, 2018.

Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. Reactnet: Towards precise binary neural network with generalized activation functions. In *European conference on computer vision*, pp. 143–159. Springer, 2020.

Zechun Liu, Zhiqiang Shen, Shichao Li, Koen Helwegen, Dong Huang, and Kwang-Ting Cheng. How do adam and training strategies help bnns optimization. In *International Conference on Machine Learning*, pp. 6936–6946. PMLR, 2021a.

Zechun Liu, Zhiqiang Shen, Shichao Li, Koen Helwegen, Dong Huang, and Kwang-Ting Cheng. How do adam and training strategies help bnns optimization? In *International Conference on Machine Learning*. PMLR, 2021b.

Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. Training binary neural networks with real-to-binary convolutions. *ICLR*, 2020.

Hai Phan, Yihui He, Marios Savvides, Zhiqiang Shen, et al. Mobinet: A mobile binary network for image classification. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 3453–3462, 2020a.

Hai Phan, Zechun Liu, Dang Huynh, Marios Savvides, Kwang-Ting Cheng, and Zhiqiang Shen. Binarizing mobilenet via evolution-based searching. In *CVPR*, 2020b.

John G Proakis. *Digital signal processing: principles algorithms and applications*. Pearson Education India, 2001.

Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, and Jingkuan Song. Forward and backward information retention for accurate binary neural networks. In *CVPR*, 2020.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pp. 525–542. Springer, 2016.

Cuauhtemoc Daniel Suarez-Ramirez, Miguel Gonzalez-Mendoza, Leonardo Chang, Gilberto Ochoa-Ruiz, and Mario Alberto Duran-Vega. A bop and beyond: a second order optimizer for binarized neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1273–1281, 2021.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, 2013.

Ziwei Wang, Jiwen Lu, Chenxin Tao, Jie Zhou, and Qi Tian. Learning channel-wise interactions for binary convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 568–577, 2019.

Zihan Xu, Mingbao Lin, Jianzhuang Liu, Jie Chen, Ling Shao, Yue Gao, Yonghong Tian, and Rongrong Ji. Recu: Reviving the dead weights in binary neural networks. *ICCV*, 2021.

Yichi Zhang, Zhiru Zhang, and Lukasz Lew. Pokebnn: A binary pursuit of lightweight accuracy. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12475–12485, 2022.

Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, 2016.