DELFT UNIVERSITY OF TECHNOLOGY

MASTER THESIS

Understanding weight-magnitude hyperparameters in training binary networks

Author:
Joris QUIST

Supervisor:
Dr. Jan van GEMERT

Daily supervisor:
Yunqiang LI

A thesis submitted in fulfillment of the requirements for the degree of Computer Science

in the

Pattern Recognition & Bioinformatics Section
Intelligent Systems

Student number: 4448979
Thesis committee: Dr. Jan van Gemert, TU Delft
Dr. Christoph Lofi, TU Delft

An electronic version of this thesis is available at
https://repository.tudelft.nl/.

October 28, 2022

TU Delft

Chapter 1

Introduction

In recent years Deep Learning has brought advances to a number of fields including computer vision, machine translation, natural language processing, and generative models. With more data and larger models, more complex tasks could be tackled. Especially in the field of computer vision the introduction of deep convolutional models has been ground-breaking. For instance, the winner of ImageNet challenge has increased the classification accuracy from 84.7% in 2012 (AlexNet [5]) to 97.8% in 2017 (SENet [4]). However, with the increase in size of these models, the computational requirements also grew. Nowadays the state-of-the-art (SOTA) networks require training on high performance computing clusters with GPUs. Even at inference time the computational and memory requirements are significant. This often makes them unsuitable to be deployed to computational-constrained hardware like mobile and embedded devices.

Currently, this is solved by sending the input from the smaller devices to a model running on remote servers with powerful GPUs which then compute and send back the results. However, this approach has some downsides. It introduces latency which can be a problem for real-time applications and it raises privacy and security concerns, since it could be personal data that is being sent back and forth.

There are many different strategies to try and reduce the requirements to run these models, without reducing their performance. In this thesis we focus on one of these strategies called Binary Neural Networks (BNNs). In a standard neural network, the weights and inputs are real-valued numbers, often represented by 32-bit floats. In a binary neural network these numbers are replaced by 1-bit numbers that can only represent $+1$ and -1. The advantages of BNNs are twofold. First the amount of memory needed to run the model is reduced by a factor of 32. Secondly, the computationally expensive multiplication and addition operations needed in standard networks can be performed using more efficient XNOR and bitcount operations when only using binary values. With modern 64-bit CPU's we can perform 64 of these binary operations for every floating-point operation, making linear and convolutional layers 64 times faster. Since often the first and last layers of the network still use real-valued weights and inputs and between the binary layers some real-valued computations happen, like batch normalization for example, the total speedup of the network is slightly lower than that, but can be up to 58 times, depending on the exact network architecture [9].

Clearly, all of these benefits, only matter if the binary networks still provide good results. It would seem that using only a single bit instead of a 32-bit float would degrade performance a lot since the network can store much less information. In practice this does not seem to necessarily be a problem. The first network with both binary weights and activations already achieved similar performance to the real-valued counterpart on small datasets like CIFAR-10 and MNIST [2]. However, on the much more difficult Imagenet classification problem, the performance dropped 29.2 percent point initially. However, a few years later with improved optimization strategies and specialized architectures now the performance drop on Imagenet is only 1.9 percent point. This is a reasonable trade off for the benefits that binary neural networks bring.

While the performance of BNNs keeps improving with new methods, the understanding of the optimization process is still lacking. Binary neural networks, by default, can not be optimized in the same way as real-valued networks. This is because gradient descent relies on making small updates to the weights to slowly improve over time. With binary weights this is not possible because the only option is to keep the same value or switch to the other value at each update. To solve this and make binary networks compatible with the standard optimization techniques, the concept of latent weights was introduced. This means that each binary weight has a corresponding real-valued latent weight. This latent weight is then updated by gradient descent using the gradient from the binary weight and the binary weight is always calculated by taking the sign of the latent weight. This simple trick makes it possible to keep training BNNs using all of the same optimization algorithms that are used for real-valued neural networks. The results show that this works well in practice, but the concept of latent weights has a few issues with it. On the one hand it is convenient to keep the same optimizers with the same hyperparameters as for real-valued neural networks, but it ignores the fact that these hyperparameters can have different effects on binary weights compared to real-valued weights. Specifically the hyperparameters that influence the magnitude of the latent weights have substantially different behaviour. This happens because the magnitude of the latent weight no longer influences the corresponding binary weight, only the sign of the latent weight matters.

This brings us to the main research question of this thesis: What is the role of weight-magnitude hyperparameters in training binary neural networks. The hyperparameters we specifically focused on are learning rate, learning rate decay, weight decay and weight initialization. We show that in the context of BNN optimization, SGD with momentum and weight decay can be interpreted using a gradient filtering perspective. This perspective offers a simplified setting with less hyperparameters to tune in total, while also giving a better explanation to these hyperparameters.

The rest of this thesis is structured as follows: chapter 2 contains a scientific article named "Understanding weight-magnitude hyperparameters in training binary networks", describing the main research that has been performed. Followed by that is supplement material, that gives background information on the different topics of the main article. These topics are deep learning in general, specific background on what binary neural networks are and how they work, and lastly a small introduction in the field of Digital signal processing, specifically on Infinite Impulse Response filters.

Chapter 2

Scientific Article

UNDERSTANDING WEIGHT-MAGNITUDE HYPERPARAMETERS IN TRAINING BINARY NETWORKS

Joris Quist, Yunqiang Li, Jan van Gemert
Computer Vision Lab
Delft University of technology
Delft, 2628 XE, The Netherlands
J.J.R.Quist@student.tudelft.nl
{Y.Li-19, j.c.vangemert}@tudelft.nl

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 discretised 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,

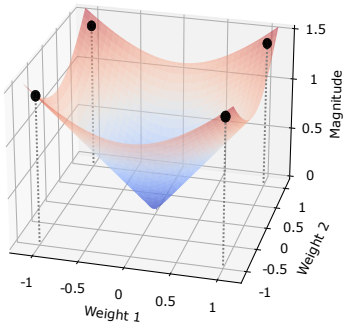


Figure 1: The magnitude for real-valued and binary weights. Changes in the real-valued weights change their magnitude. For binary weights, however, the magnitude will never change. Thus, magnitude-based hyperparameters need reinterpretation for binary weights.

learning rate, learning rate decay, and weight decay. 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.

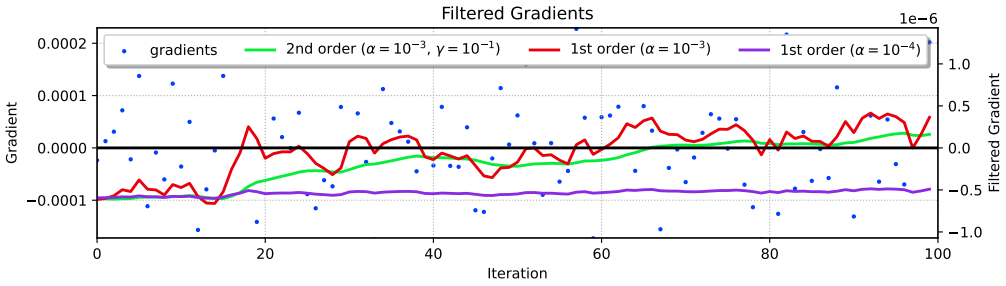
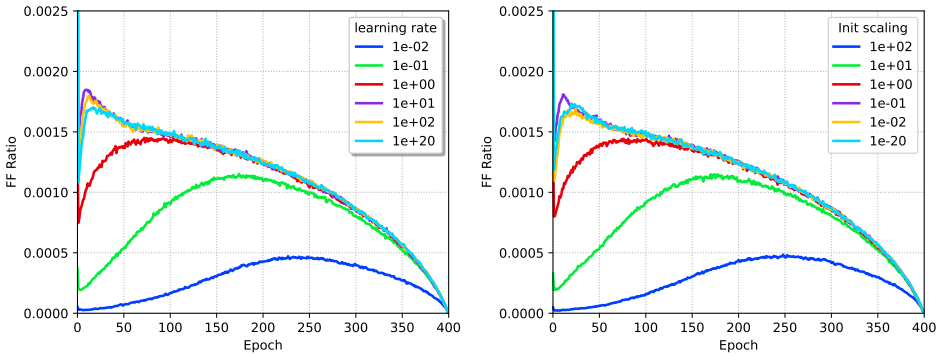


Figure 2: Gradients and filtered gradients using a first and second order filter for a single epoch of training on CIFAR-10. For better visualisation, the filter outputs are scaled up to a similar range as the unfiltered gradients. It can be seen that the unfiltered gradients are noisy and that the filtered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

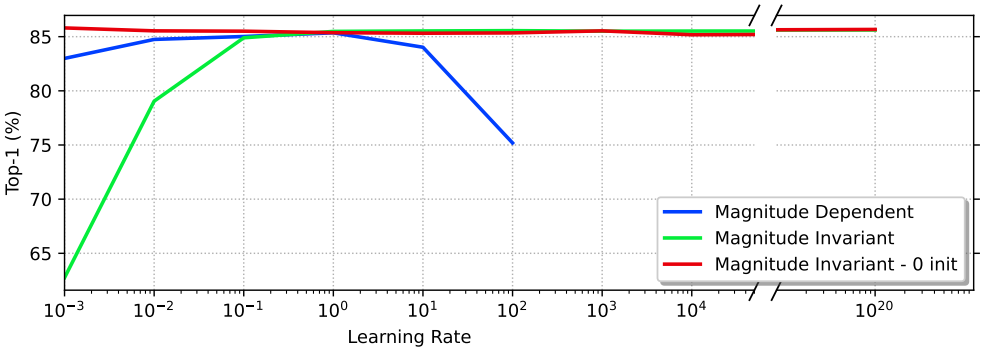


Figure 4: Learning rate effect on accuracy for three settings. (a). standard SGD. (b) Magnitude invariant in Eq 3, by removing clipping and scaling. (c) Magnitude invariant with latent weight initializes of 0 in Eq 8. Setting (a) has just a single optimum. The accuracy in setting (b) is sensitive to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

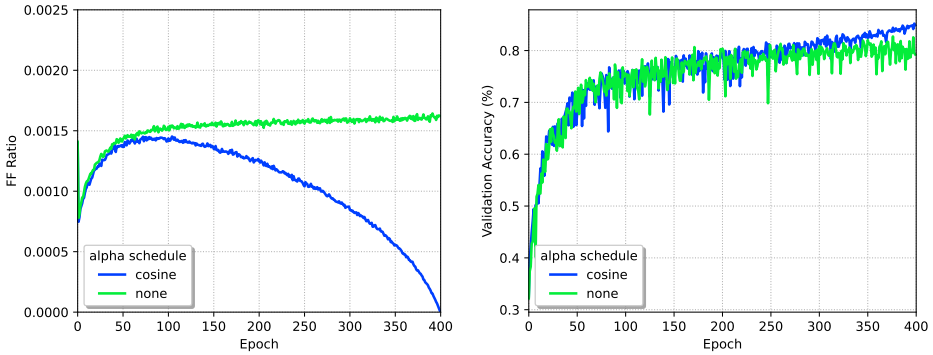


Figure 5: Flipping rate and accuracy for comparing alpha decay vs no alpha decay. When using cosine alpha decay the FF ratio goes towards zero over time while without decay, the flipping will continue, causing the network to not fully converge leading to reduced accuracy.

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

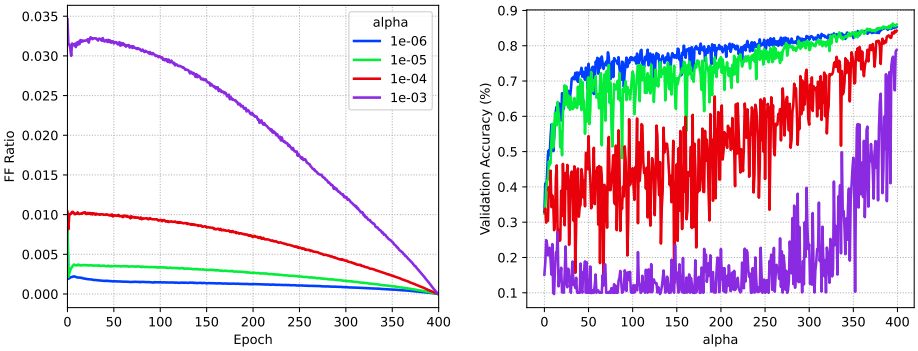


Figure 6: FF ratio and accuracy for varying alphas. BNNs are sensitive to alpha, similar to how they are sensitive to weight decay, because it has a big influence on the FF ratio, which really affects performance. A too high FF ratio leads to noisy training and poor results in the first half of training, 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

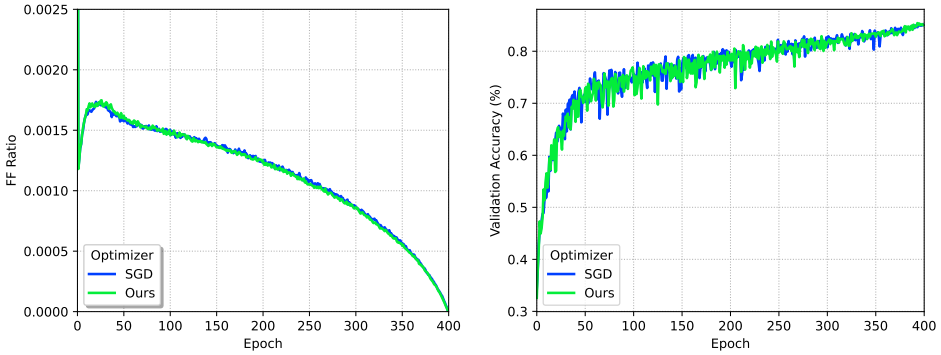


Figure 7: Flipping rate and accuracy for SGD with latent weights in Eq 3 and our gradient-filtering optimizer without latent weights in Eq 18. The methods are empirically equivalent.

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 * 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.

Chapter 3

Supplement Materials

3.1 Deep Learning

This section will give an short introduction on deep learning. Deep learning is a specific subset of methods within machine learning. Before deep learning, the strategy was often to first do feature extraction on the input using hand-made algorithms. Deep learning is based on deep neural networks, which means that there are many layers in the network, where the first layers are then responsible for learning the features. This eliminates the need for specific domain knowledge which is needed to extract good features. Because the features are learned during training, it learns to extract features that maximise performance. The first part of this section explains what a neural network is and how it works. To make the neural networks learn it needs an optimization algorithm, also called optimizers, the second part explains more about how these work and what different kinds of optimizers there are.

3.1.1 Neural Networks

The neural networks used in deep learning are specifically artificial neural networks. This is because they were originally inspired by actual biological neurons, although since then this relationship with real-life neurons has decreased over time. Neural networks consist of layers of neurons, where each layer gets inputs from the previous layer and passes the outputs on to the next layer. The output of each neuron in the layer is equal to a weighted sum over its inputs, followed by some non-linear operation, called the activation function. Originally the inputs of each neuron were all the outputs of the previous layer, these networks were therefore called fully-connected neural networks. The way a network calculates its output is by giving the input to the first layer, which calculates its output, which is again the input for the next layer. This continues through all the layers until the overall output of the network is calculated by the final layer. This process is called forward propagation. The weights for the weighted sum are called the parameters of the neural network and these are the values that are being optimized during training. The way this works is using training examples. These examples are input-output pairs that describe the problem and are used to teach the network what output it should generate for a certain input. At first the weights are randomly initialized, so the outputs will be very different from the target outputs. To update the weights first a loss function is used to calculate how far off the network was. A popular example of this is mean squared error, where the error is the difference between the output and the target output. The goal is then to minimize this loss. Since there is no closed-form solution to compute the weights for which the loss is minimal, this needs to happen iteratively. To do this the gradient over the loss function with respect to the weights can be calculated. Because of the way these networks are structured, the gradients can be efficiently calculated layer by layer, using the chain rule. It starts at the back of the network were the loss is calculated and the gradient than propagates backward through the network, which is called back-propagation. The gradients over the weights can then be used in a process called gradient descent to find a local minimum

for the loss function. This learning process is then repeated for a fixed number of steps or until the network no longer improves. To confirm that the resulting network also works well for data that it has not seen yet, a set of input-output pairs that were not used during training, is used to validate the performance.

3.1.2 Optimizers

A very important aspect of deep learning are the optimization algorithms, often just referred to as optimizers, since these actually update the weights and are what makes the network learn. These algorithms are based on the gradient descent algorithm. In this subsection we will explain what gradient descent is and its shortcomings which the other optimizers try to solve.

Gradient descent Gradient descent is an optimization algorithm that tries to find local minima of differentiable functions. It works by iteratively updating an input. First an initial input or guess is given, then the derivative over the function is computed with respect to this input. This gradient is then used to take small step in the opposite direction to get a new input which, if the step is small enough, will lead to a smaller output. This process is then repeated until the algorithm has converged. This looks as follows:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta \delta F(\boldsymbol{x}_t) \tag{3.1}$$

Here $\boldsymbol{x}$ is the input, η is a scaling factor that determines the size of the update step and is often called the learning rate and F is the function that is being optimized.

This algorithm can be applied to optimizing neural networks. This works by creating a loss function that is used as the function that is being minimized. This loss is computed between the outputs of the network of an input dataset and the corresponding ground-truth outputs in the dataset. This can for example be the mean squared error (MSE) which is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{3.2}$$

So in this case Y_i are the outputs generated by the network and $\hat{Y}_i$ are the ground-truth labels in the dataset.

The input of the gradient descent algorithm are the weights of the neural network. So the derivative with respect to the weights of the loss function is calculated and is used to update the weights. Again, this process is repeated for a fixed number of iterations or until the algorithm has converged. In this setting the function looks as follows:

$$w_t^i = w_{t-1}^i - \eta \nabla_{w_{t-1}^i} \tag{3.3}$$

Where w_t^i is the weight with index i at time step t, from here on we omit the index for clarity. $\nabla_{w_{t-1}^i}$ is the gradient over the loss function with respect to the weight w at time step t. The learning rate is still η and is a very important parameter for gradient descent. If its too small it means that all the steps the algorithm takes are small and it could mean that the minimum is never reached. If its too big it means that the optimizer may overshoot the minimum and actually end up worse at every update, which also means a minimum is never reached. To get good results the learning rate needs to be somewhere at a sweet spot in between.

Gradient descent is a very powerful concept that can be applied to many problems, but this vanilla version of the algorithm has a few problems that make it unsuitable for use in deep learning. The main problem is that compute the gradient and do an update step the network has to be evaluated on the entire dataset, but still only small steps can be taken for the algorithm to work. This is very slow, especially since deep learning works by using large amounts of data. There will also be many similar items in the dataset which mean that a lot of redundant computation is happening.

The second big problem is that gradient descent finds local minima. For convex functions this is not a problem, since there is only one minimum, which is the global minimum. However, the loss functions used in deep learning are non-convex and very noisy. This means that there are many local minima, many of which are not good solutions to the problem. Gradient descent will often converge to the minimum located in the valley where the optimization starts, so the end result is strongly influenced by the initial randomly generated weights.

Stochastic gradient descent Both of these problems can be solved with stochastic gradient descent (SGD). Instead of calculating the loss for the entire dataset at once, it will calculate the loss for each element in the dataset one by one and each time calculate the corresponding gradient and parameter updates. This means that the algorithm runs much faster, since there are many more updates to the parameters compared to only updating them each epoch. Since the loss is also calculated with different training examples every time, the loss landscape also is different every time. This makes it possible for the algorithm to escape local minima, because the locations of these minima changes.

A disadvantage of this method is that using only a single training example is that the gradients and therefore the training process can become very noisy. At every update the training algorithm could think that the minimum is somewhere else, which can make the updates go back and forth without actually making progress. To reduce the noise, what is actually mostly used is in practice mini-batch gradient descent. This means that instead of a single training sample, a mini-batch of multiple training samples is used. A larger sample size reduces the randomness and therefore also reduces the noise.

Momentum Another method to reduce the effect of negative effect of noisy gradients is to use momentum. This means that instead of using only the gradient of the current batch, a history of past gradients is used. This history is calculated by using an exponential moving average (EMA). An EMA is a specific version of an Infinite Impulse Response filter (Section 3.3). It is not desirable to calculate the average over all past gradients, because the older the gradient is the less relevant it is right now. That is why a moving average is used, which means that the average is mostly based on more recent inputs. A simple example of a moving average is a sliding windows moving average, which is calculated by simple calculating the average over a window of the last n inputs. This is not suitable for the use case of gradients, because there is a gradient for every weight in the neural networks, which would mean that for each weight in the network the n past gradients would need to be remembered, which costs a lot of memory for these large deep neural networks. This is where the exponential moving average comes in. Instead looking at past inputs, it looks at the previous output. It is calculated as follows: $y_i = (1-\gamma) \cdot y_{i-1} + \gamma x_i$, where y_i is the current output, y_{i-1} is the previous output, x_i is the current input and $(1-\gamma)$ is the discount factor. The discount factor determines how much past inputs weigh versus the current input. A high discount factor means that the moving average more quickly “forgets" past inputs, while a low discount factor means past values have an influence that lasts longer. The advantage of this is that only one value needs to be remembered which is the previous output and this saves memory. The exponential part is that the effect of previous inputs decays exponentially, because at

every step the previous inputs are multiplied with $(1-\gamma)$. The optimization algorithm can then be described as:

$$m_t = (1-\gamma)m_{t-1} + \gamma\nabla_{wt-1} \tag{3.4}$$
$$w_t = w_{t-1} - \eta m_t \tag{3.5}$$

Using this smoothed gradient instead of the original gradient means that the signal is more consistent and means the optimizer is less likely to focus to much on noise that is not beneficial reaching a good performance.

Adam The last optimization algorithm we will discuss is Adam. The name Adam comes from adaptive moment estimation. It still is based on batch-gradient descent and also uses momentum, but it also introduces an adaptive learning rate, based on the magnitude of the gradients. The idea behind is that it often happens that the magnitude of gradients for different weights can be very different. This means that the optimizer takes big steps for some weights while only taking small steps for others. This makes it difficult or sometimes even impossible to find a good learning rate. By computing a second EMA, but this time not over the gradient but over the squared gradient, the magnitude of the gradients for each weight can be estimated by taking the square root of the average squared gradient. Together these EMAs look as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_{\theta_t} \tag{3.6}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla^2_{\theta_t} \tag{3.7}$$

Here the discount factor $(1-\gamma)$ has been replaced by β_1 and β_2 is the discount factor for the so called second order momentum. The weight is then updated with:

$$w_t = w_{t-1} - \eta\frac{m_t}{\sqrt{v_t}+\epsilon} \tag{3.8}$$

The epsilon here is a small value that is added to the denominator to prevent dividing by zero.

3.2 Binary Neural Networks

This section will give some background information on what binary neural networks are exactly. This is done by discussing two of the first works in the field: BinaryNet[2] and XNOR-Net[9]. The methods and strategies proposed on these works are still used in more recent works.

3.2.1 BinaryNet

While it was not strictly the first work on BNNs, the BinaryNet algorithm [2] is what marked the real beginning of the binary neural network field and has been used a basis for training BNNs every since. The main problem with BNNs was that the process of first training a full precision network and only quantizing at the end of training does not work. This process does work when quantizing 32-bit floats to 8-bit integers By strategically choosing which real valued numbers are mapped to which integers, the difference in performance can stay small. This is because the quantization error, which is the error between the real-valued weight and

the quantized weight, stays relatively small. For binary neural networks this is not the case. The only two options for weights are -1 and $+1$. When trying to minimize the quantization error it is best to map all the positive numbers to $+1$ and all the negative numbers to -1. This function is called the sign function, because it only looks at the sign of the number.

$$\text{Sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{otherwise.} \end{cases} \tag{3.9}$$

This will give a big quantization error. In practice this means when it is applied to a real valued network, the difference between the real-valued and binary weights is so big, the performance will drop to the same level as before training. The creators of BinaryNet realized that what was necessary is for the binary neural network to be trained to be trained directly so that the optimization process can take the binarization into account.

However, it is not possible by default to use standard neural network optimization techniques. Neural networks are mostly optimized using different variation of gradient descent. Gradient descent works by making many small incremental updates to real-valued weights. This is not possible when using binary weights because they are discrete, so it is not possible to make these small incremental updates. The only possibility is to keep the same value as the previous step or invert it.

To get around this problem and make BNNs compatible with gradient descent algorithms, latent weights were introduced. Each binary weight has its own real-valued latent-weight. This weight can be incrementally updated. The training algorithm was then updated to optimize the latent weights but still function as a binary network. This works as follows. For each binary weight in the network a corresponding latent weight is created. During the forward pass a sign function is applied to the latent weights to generate the binary weights. The sign function is used because this leads to the smallest quantization error between the latent and binary weights. The binary weights are then used as normal in convolutional and linear operations. During the backward pass first the gradients with respect to the binary weights are calculated:

$$g_b = \frac{\delta \mathcal{L}}{\delta b} \tag{3.10}$$

Where b is the binary weight, $\mathcal{L}$ is the loss and g_b is the gradient with respect to the binary weight. Then the gradient with respect to the latent weight can be calculated as follows using the chain rule:

$$g_w = \frac{\delta \mathcal{L}}{\delta w} = g_b \cdot \text{sign}'(w) \tag{3.11}$$

Where w is the latent weight and g_w is the gradient with respect to the latent weight. However, this introduces another problem. The derivative of the sign function is zero everywhere, except at zero where it is undefined. This means that the latent weights do not have any proper gradients and without any proper gradients the latent weights can not be updated, so training is not yet possible.

This is where the straight-through-estimator (STE) comes in. In its simplest form the STE will simply pretend during back propagation that the binarization never happened, but instead the identity function was applied. This means that it approximates the gradient of the sign function with that of the identity function.

$$g_w = g_b \cdot \text{sign}'(w) \approx g_b \cdot \text{identity}'(w) = g_b \cdot 1 = g_b \tag{3.12}$$

Now that the gradients can be calculated for the latent weights they can be updated. To also get binary activations, the sign function is used as the non-linear activation function. This again gives the same problem as with the weights, that the gradient is not propagated through the sign function and in this case there would still be no gradients anywhere in the network except in the last layer. The solution is again to use the Straight-through-estimator to bypass the sign function on the backward pass. Then gradients can flow back through the network again.

Now all the steps are there to be able to use gradient descent to train binary neural networks. They do add one last thing to the weight update step, which is weight clipping. After the update all of the latent weights are clipped to lie within the range [-1, +1]. This makes sure that the latent weights will not infinitely grow larger without actually influencing the binary weight.

BinaryNet performed quite well on smaller datasets, on CIFAR-10 using the VGG-Small architecture[10] it reached 91.7% accuracy which is only 2.1 percent-point less than the 93.8% accuracy its real-valued counterpart achieved. However, its ImageNet performance showed a different picture. Where the real-valued AlexNet[6] reaches 57.1% top-1 accuracy the BinaryNet version only reaches 27.9%. This is a drop of 29.2 percent-point, more than halving the original performance. It was speculated that this performance degradation was an inherent result of the binarization process because of the representational capabilities of binary networks. However, lots of research since then has been spent on closing this gap between real-valued and binary networks. The first step was taken by the introduction of XNOR-Net, which to this day is still the foundation for most work on BNNs.

3.2.2 XNOR-Net

The contributions in this work are two-fold. The first is improving the representational capabilities of binary networks by introducing real-valued channel-wise scaling factors for the outputs of the convolutional layers. The second is the reduce the loss of information flowing through the network by reordering the structure of blocks in the network.

Scaling The authors argue for an approximation viewpoint of BNNs. The real-valued original weights are approximated by binary weights such that: $\boldsymbol{W} \approx \boldsymbol{B}$, where $\boldsymbol{W} \in \mathbb{R}^{c \times w \times h}$ and $\boldsymbol{B} \in \{-1, +1\}^{c \times w \times h}$ are both convolutional filters. To improve approximation accuracy and thus reduce the quantization error, a real-valued scaling factor α is introduced, so that the approximation becomes: $\boldsymbol{W} \approx \alpha \boldsymbol{B}$. The optimal values for α and $\boldsymbol{B}$ can be found by solving:

$$J(\boldsymbol{B}, \alpha) = ||\boldsymbol{W} - \alpha \boldsymbol{B}||^2 \tag{3.13}$$

$$\alpha^*, \boldsymbol{B}^* = \arg\min_{\alpha \boldsymbol{B}} J(\boldsymbol{B}, \alpha) \tag{3.14}$$

The solution to this minimization problem is for the binary weights to still be calculated by taking the sign of the latent weights $\boldsymbol{B} = \text{sign}(\boldsymbol{W})$ and for α to be the mean absolute value of the latent weights, so $\alpha = \frac{1}{n}||\boldsymbol{W}_{l1}||$. Convolutions can then be approximated using $\boldsymbol{I} \circledast \boldsymbol{W} \approx (\boldsymbol{I} \circledast \boldsymbol{B})\alpha$, where $\boldsymbol{I}$ are the activations and $\circledast$ is the convolutional operator. Note that the scaling factor is applied after the convolution to make sure the convolution can be computed with the binary weights.

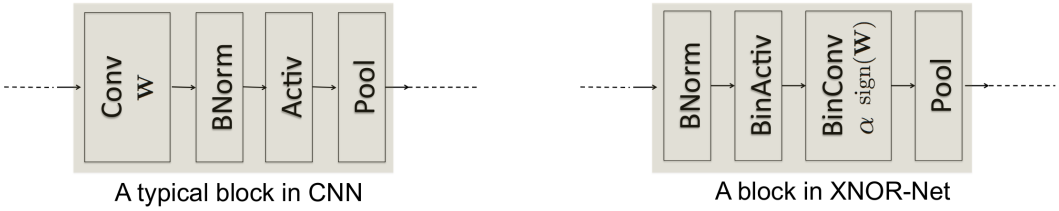


FIGURE 3.1: This figure shows the original block structure of CNNs on the left side, consisting of first a convolutional layer, then a batch normalization layer, followed by the activation function and lastly an optional pooling layer. The new structure on shown on the right. The block now starts with batch normalization and the binary activation before the binary convolution and then still the pooling layer as last, to prevent information loss when pooling binary values. (Image taken from [9])

To be able to use the more efficient XNOR and bitcount operations for convolutions the activations have to binarized as well. A similar strategy as with the weights was applied here. Again to decrease the quantization error a real-valued scaling factor was introduced for the activations. However, instead of one value per output channel there is one value for every possible sub-tensor in the input with the same size as the weight filters. This is equivalent to on scaling factor for every "pixel" in the output, so $\boldsymbol{K} \in \mathbb{R}^{w_{out} \times h_{out}}$, where w_{out} and h_{out} are the output width and height respectively. The scaling factors are calculated by taking the average of each of these sub-tensors in the input. While for the weights the scaling factors can be calculated during training and stay fixed afterwards, the inputs change each forwards pass so the input scaling factors have to be recalculated each time. This does add some additional overhead, but is still relatively small compared to the amount of operations for a convolution operation, even if it is an efficient binary convolutions. However, because of this increased overhead, while also in practice giving relatively little benefit over the weight-based scaling, subsequent works, mostly use just the weight-based scaling factor.

Block structure The second contribution of XNOR-Net is a new block structure for the layers in the network. The original BNN uses the same blocks as standard CNNs, which means that the now binary activation function comes right before the pooling layer. However, this causes a large loss of information. For example, when using the popular max-pooling on the binarized values, almost all outputs will be equal to +1, since to be -1, all input values need to be -1. The new block architecture is shown in 3.1.

Together these two contributions result in XNOR-Net reaching an accuracy of 44.2% on Imagenet with the AlexNet architecture. This is an increase of 16.3 percent point over the original BNN, closing the gap to the full-precision AlexNet to only 12.4 percent point. This made binary networks a much more viable solution than was previously thought and sparked the takeoff of the BNN field.

3.2.3 Efficiency

In this section we will discuss the efficiency aspect of binary neural networks. As shown in the previous chapter, there still is some performance degradation compared to real valued networks. A lot of works try to reduce this gap, but this is only worth the effort because the resulting networks are actually more efficient. Where does this efficiency come from? Binary convolutions and binary matrix multiplications can be made more efficient because they consist of many dot-products which can be computed using XNOR and bitcount operations

Multiplication				XNOR		
$input_1$	$input_2$	output		$input_1$	$input_2$	output
+1	+1	+1		1	1	1
+1	-1	-1		1	0	0
-1	+1	-1		0	1	0
-1	-1	+1		0	0	1

TABLE 3.1: This table shows the relationship between multiplying plus and minus ones and the XNOR operation. You can see that representing +1 with 1 and -1 with 0, results in the XNOR operation exactly represents the original multiplication.

when both the input and weights are binary values representing +1 and -1. This works as follows, the dot product consists of two steps: multiplication and addition.

For the multiplication the output is always +1 if both inputs are the same and -1 if both inputs are the opposite. For binary values the +1 is represented by a 1 and the -1 is represented by a 0. So, the output of multiplying two binary inputs is the same as applying the XNOR operation on their representations, because if the two inputs are the same the output will be one, which maps to +1 and if both values are different the output will be 0 which maps to -1.

Additions can be performed using the bitcount operation. The bitcount operations basically does what the name implies. It takes n bits as inputs and gives as output the number of bits that are 1 as n. n here is not the same answer to summing the actual +1/-1 weights, because the zeros are ignored. To get the actual sum s the amount of zeros need to be calculated and subtracted from the answer which gives: $s = c - (n - c) = 2c - n$.

On modern 64-bit CPUs both the XNOR and bitcount operations can be performed for 64 bits at the same time, while for floating-point operations only one can be done at the same time on a single core. This is were the often mentioned 64x speed improvement for BNNs comes from. The exact speedup is however also very dependent on the hardware used. For example, the total amount of operations that can be achieved on modern computers is often bottle-necked by the speed by which the data can be loaded into the correct registers. So while BNNs are definitely faster and more efficient for the hardware, there is no speedup factor that always holds.

3.2.4 Improvements

This section discusses some of the works from the last few years that improve on the baseline that has been set by XNOR-Net. The strategies to achieve these improvement can be divided into three categories. These are: introducing real-valued scaling factors, improving the loss function and reducing the gradient error.

Reducing Quantization Error The idea behind reducing the quantization error is that reducing the error between the real-valued and binary convolution, also leads to a reduction in the performance gap between the two. This is the idea behind the real-valued scaling factors introduced in XNOR-Net.

Introducing real-valued scaling factors Since binary neural networks have a more limited representational power, compared to real-valued networks, one popular method to improve their performance is to introduce real-valued scaling factors. This is done so that the binary convolutions can better approximate the real-valued ones and by doing it in strategic places,

it only adds a small amount of extra operations even compared to the efficient binary convolutions. The best and most used example of this is the analytically calculated scaling factor from XNOR-Net[9], explained earlier, but later more techniques have been proposed

As a direct successor to XNOR-Net, XNOR-Net++ [1] was proposed. It argues that the analytically calculated scaling factors for both the weights and the activations are quite expensive to compute. For the activations this is relatively expensive because the scaling factor needs to be calculated for every input and has thus to be performed on inference time. In this work, a new method is proposed that merges both scaling factors into one new scaling factor Γ, which is not analytically calculated but instead trained together with the weights via backpropagation. This makes them more flexible, since they can be trained like any other parameter and has the advantage no scaling factors need to be computed at inference time. Four different ways to methods to construct Γ are proposed. The first one is a simple case where there is a scaling factor for each channel in the output:

$$\Gamma = \alpha, \alpha \in \mathbb{R}^o \tag{3.15}$$

Where o is the number of output channels. Case 2 is a much more flexible option where there is one scaling factor for each pixel in the output:

$$\Gamma = \alpha, \alpha \in \mathbb{R}^{o \times h_{out} \times w_{out}} \tag{3.16}$$

Where h_{out} and w_{out} are the height and width of the output respectively. This works better than case 1 because it also can take statistics in the spatial dimensions into account. However, since it is more flexible it can also lead to more overfitting. To reduce the flexibility but still take the spatial dimensions into account, in case 3 and 4 are decompositions of case 2. In case 3 there is a decomposition into two terms: one for the spatial dimensions and one for the output channel dimensions:

$$\Gamma = \alpha \otimes \beta, \alpha \in \mathbb{R}^o, \beta \in \mathbb{R}^{h_{out} \times w_{out}} \tag{3.17}$$

This reduces the amount of trainable parameters from $o \times h_{out} \times w_{out}$ to $o + h_{out} \times w_{out}$, but actually increases the performance by 0.6 percent-point accuracy on Imagenet with ResNet-18 compared to the second case. Then for the fourth case with the decomposition in all dimensions:

$$\Gamma = \alpha \otimes \beta \otimes \gamma, \alpha \in \mathbb{R}^o, \beta \in \mathbb{R}^{h_{out}}, \gamma \in \mathbb{R}^{w_{out}} \tag{3.18}$$

This reduces the amount of parameters even further to $o + h_{out} + w_{out}$, but again increase performance by 0.4 percent point compared to the third case.

This is similar to the approach proposed by Tang et al. [11]. They introduce a scaling factor that is learned as well, but in a different way. They replace the standard ReLU action function with the PReLU activation function. Where the ReLU function is fixed to $\text{ReLU}(x) = max(0, x)$, PReLU instead has an extra scaling factor for when x is smaller than 0: $\text{PReLU}(x) = \max(0, x) + a \cdot \min(0, x)$.

A quite different method was introduced by Martinez et al.[8] They argue that the previous scaling factor are more limited, because even tough they are learnable parameters, after training they are fixed. Instead, they propose a new method named data-driven channel re-scaling. This means that they take the activations into account and use that to do a channel-wise scaling of the convolutional layer outputs. Figure 3.2 shows the used architecture for this. This method is much more flexible than the previous ones, because it can adapt to different inputs, while still only introducing few extra computations.

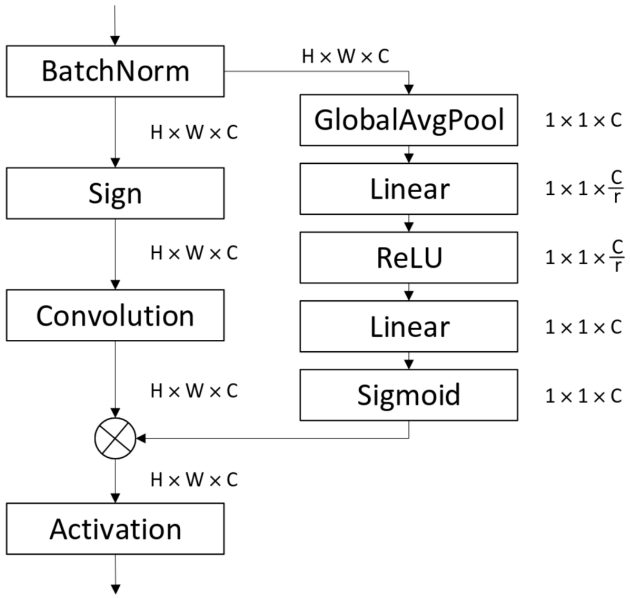


FIGURE 3.2: This shows that architecture for a block in Real-to-Binary Net[8]. The left-hand side shows the standard structure for BNNs and the right-hand side shows the new gating function that computes the scaling factors. Here H, W C are the height, width and amount of channels of the input respectively. To make the whole function more efficient it starts with global average pooling to only keep the channel dimension. The first linear layer than also scales down the amount of channels by a factor r to reduce the amount of needed computation even further. (Image taken from [8])

Loss Functions

Another area of focus for improving the performance of BNNs is in the loss function that is being used to train the networks. This is also what "Real-to-Binary Attention Matching" is about, introduced as another contribution in Martinez et al.[8]. This consists of a student-teacher model, where they train the student to match the output of the teacher, using the error between the logits produced by both. This setup is also used in general student-teacher training with full-precision neural networks. However, the novel part is that the architecture of the teacher network matches the student as closely as possible, so that each block in the student model has an accompanying block in the teacher model. At the end of each of these blocks the error between the attention maps between the student and teacher is calculated and added to the loss. These intermediary error signal help training the binary networks by preventing signal degradation during propagation of gradients through the entire network and makes sure that the output of the binary convolutions better matches the output of full-precision networks.

There can be quite big differences between the activations of binary and real-valued networks, so the authors proposed a three step training strategy to minimize these differences. First the student is still a real-valued network but with the structure of a binary network. In the next step the trained student becomes the teacher and the new student now has binary activations. In the last step the teacher is again the student from the previous step and now the student has both binary weights and activations. This way at each step there is a smaller difference between the student and teacher, which makes it easier for the student network to match the outputs of the teacher.

A subsequent work called ReActNet [7] improves upon this idea, but argue that the student does not necessarily need to exactly match the outputs of the teacher, but that the outputs

of the student should match the distribution of the output of the teacher. They do this by giving a new loss function that is defined as the KL divergence between the softmax output of the real-valued network and a binary network. They found that with this new loss they did not need to match the output of every block in each of the networks, but only that of the final output of the networks. This makes the training process easier and more flexible, because the architecture of the teacher network does not necessarily need to match the architecture of the student network.

3.2.5 Without Latent Weights

All the previously mentioned works on BNNs, build upon the concept of latent weights. Helwegen et al. [3] are the first to provide a new perspective that is not based upon latent weights. The perspective that is linked to using latent weights is to see the binary weights that are computed are an approximation of the real-valued latent weights. They argue that this perspective is problematic. After training a BNN, if you evaluate the network using the latent weights instead of the binary weights, the resulting accuracy are worse than when you would evaluate the binary weights. It therefore, does not make sense to say that binary weights are approximating latent weights if they perform better than the thing that they are approximating.

Instead the propose a new inertia-based view, where they give another explanation for why training BNNs with real-valued latent weights work. They argue that the role of the latent weights is twofold. The sign of the latent weight encodes the value for the binary weight, while the magnitude of the latent weight is used as inertia for the network. It makes sure that even if the gradient signal is noisy and keeps switching signs, the corresponding binary weight remains more stable. If the magnitude of the latent weight is big, it takes a larger and more consistent gradient in the opposite direction to flip the binary weight.

To make these different roles more explicit, they propose a new Binary Optimizer (BOP). This optimizer does not optimize latent weights, but directly operates on the binary values. The way this optimizer works is uses an exponential moving average over the gradients to provide the inertia:

$$m_t = (1-\gamma)m_{t-1} + \gamma g_t = \gamma \sum_{r=0}^{t} (1-\gamma)^{t-r} g_r \tag{3.19}$$

This is equivalent to the momentum term in SGD, but it has a different role, because it replaces the latent weights to provide inertia. With an important detail being that this tracks the gradient instead of the negative gradient like latent weights. Since an optimizer that works directly can only choose to flip a weight or keep it the same, they make this explicit in the way the binary weight is defined:

$$b_t = \begin{cases} -b_{t-1}, & \text{if } |m_t| \geq \tau \text{and } \operatorname{sign}(m_t) = \operatorname{sign}(b_{t-1}) \\ b_{t-1}, & \text{otherwise} \end{cases} \tag{3.20}$$

Here b_t is the binary weight at time step t and τ is a threshold. The way it works is that binary weight should mostly have the opposite sign to the momentum term, so when their signs are the same the binary weight will flip. Unless the magnitude of the momentum term is below the threshold value. This threshold is introduced to prevent the binary weight potentially flipping rapidly when the momentum term gets close to zero.

They show that with this new optimizer without latent weights they still can still get competitive results compared to existing methods with latent weights.

3.3 Filtering

The last important concept that is relevant to the scientific article is digital signal processing (DSP). Digital signal processing is a large field that has a lot of applications, like speech processing, video coding and image compression for example. We will, however, focus only on a subset of DSP called digital filtering. A digital filter performs operations on digital signals, which in turn are sampled, discrete-time signals. The goal often is to attenuate certain frequencies from the signal. For example, audio signals can contain high frequency noise, which is undesirable. To reduce this noise a digital filter can be used to attenuate the high frequencies, while keeping the lower frequencies. Such a filter is called a low-pass filter, because the low frequencies pass through the filter unchanged.

The specific filter that is relevant to our work is the exponential moving average mentioned in the previous sections and is a very simple example of a specific type of low-pass filter, a linear Infinite Impulse Response Filter. This is important, because this means that we can treat EMAs as digital filters which allows us to use DSP analysis techniques to better understand them. The “infinite impulse response" part of the name comes from the fact if you give an impulse input to the filter, then the effect of this impulse can be measured infinitely in the response, or output, of the filter. It’s a linear filter because it is a linear combinations of past and current inputs and past outputs. The general formulation looks as follows:

$$y_t = \frac{1}{a_0}(b_0 x_t + b_1 x_{t-1} + ... + b_P x_{t-P} \tag{3.21}$$

$$-a_1 y_{t-1} - a_2 y_{t-2} - ... - a_P y_{t-Q}), \tag{3.22}$$

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. Since an EMA only looks at the current input and the previous output, so at most one step back, it is a first order filter. To show that an EMA is indeed an IIR filter we can transform it into the general form, by choosing the correct a and b vectors:

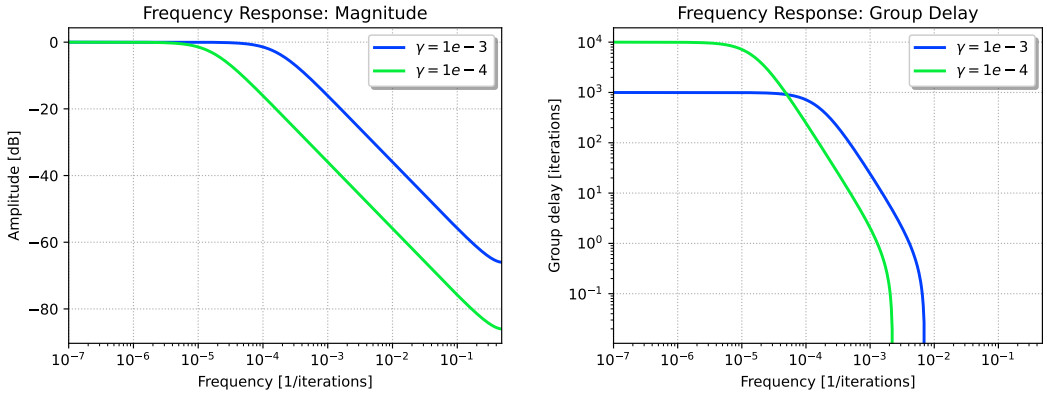
$$a = \begin{bmatrix} 1 \\ \gamma - 1 \end{bmatrix}, \quad b = \begin{bmatrix} \gamma \\ 0 \end{bmatrix} \tag{3.23}$$

$$y_t = \frac{1}{a_0}(b_0 x_t - b_1 \cdot x_{t-1} - a_1 y_{t-1}) \tag{3.24}$$

$$= \frac{1}{1}(\gamma x_t - 0 \cdot x_{t-1} - (\gamma - 1) y_{t-1}) \tag{3.25}$$

$$= (1 - \gamma) y_{t-1} + \gamma x_t \tag{3.26}$$

Knowing this, we can perform frequency analysis on EMAs, to get more insight into what effect the filter has on different frequencies. The most important question is often what frequencies the filter attenuates and how much. Another quite relevant aspect is that filters almost always introduce some delay between the inputs and outputs, even for frequencies that are not filtered. See Figure 3.3 for an example of both of these for two EMAs with different discount factors.



(A) The amplitude of response for different frequencies in decibels. When the amplitude is 0dB, it means that the frequency passes through the filter not attenuated. For every -3dB, it means that the magnitude of the frequency in the output is half of the magnitude of the frequency in the input. It can be seen for the EMA with the smaller γ the range of frequencies that are attenuated starts lower than that with the bigger γ. It also clearly shows that EMAs are low-pass filters, since the lower frequencies are untouched while the higher frequencies are attenuated.

(B) The delay that is introduced in the signal is called group delay and is expressed in a number of samples, or iterations in our case. This shows that the EMA that filters more frequencies also introduces more delay in the signal, a low-frequency signal that starts appearing in the input signal at a certain iteration can take ten thousand iterations before it shows in the output of the EMA.

FIGURE 3.3: These figures show two different types of analysis for two instances of an EMA with different discount factors γ

Bibliography

[1] Adrian Bulat and Georgios Tzimiropoulos. Xnor-net++: Improved binary neural networks, 2019. 23

[2] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016. 1, 18

[3] 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*, 32:7533–7544, 2019. 25

[4] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018. 1

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012. 1

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. 20

[7] 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*, pages 143–159. Springer, 2020. 24

[8] Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. Training binary neural networks with real-to-binary convolutions. *ICLR*, 2020. 23, 24

[9] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016. 1, 18, 21, 23

[10] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2015. 20

[11] Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In *AAAI*, pages 2625–2631, 2017. 23