Learning Discrete Weights Using the Local Reparameterization Trick

Oran Shayar*† Dan Levi * Ethan Fetaya‡

October 24, 2017

Abstract

Recent breakthroughs in computer vision make use of large deep neural networks, utilizing the substantial speedup offered by GPUs. For applications running on limited hardware however, high precision real-time processing can still be a challenge. One approach to solve this problem is learning networks with binary or ternary weights, thus removing the need to calculate multiplications and significantly reduce memory size and access. In this work we introduce LR-nets (Local reparameterization networks), a new method for training neural networks with discrete weights using stochastic parameters. We show how a simple modification to the local reparameterization trick, previously used to train Gaussian distributed weights, allows us to train discrete weights. We tested our method on MNIST, CIFAR-10 and ImageNet, achieving state-of-the-art results compared to previous binary and ternary models.

1 Introduction

Deep Neural Networks have been the main driving force behind recent advancement in machine learning, notably in computer vision application. While deep learning has become the standard approach for many tasks, applying it using low power and memory constraints on dedicated hardware is still a challenging task. This is especially challenging in autonomous driving in electric vehicles where high precision and high throughput constraints are added on top of the low power requirements.

One approach to solving this task is by learning networks with binary \{±1\} or ternary \{−1, 0, 1\} weights [1, 16] that require an order of magnitude lesser memory and no multiplications which can be done much faster on dedicated hardware. The problem arises when we try to backpropagate errors as the weights are discrete. One heuristic suggested in [1] is to use stochastic weights

* General Motors Advanced Technical Center - Israel
† Department of Electrical Engineering Technion
‡ Department of Computer Science, University of Toronto
sample binary weights \( w_b \) for forward and gradient computation and update the stochastic weights instead. Another idea used in [6, 16] is the use of the "straight-through" estimator \( \frac{\partial \text{sign}}{\partial r} = r \mathbb{1}[|r| \leq 1] \). While these ideas were able to produce good results even on reasonably large networks such as resnet-18 [5], there is still a large gap in performance between the full-precision network and the discrete networks.

In this work we try to train neural networks with discrete weights using a more principled approach. Instead of trying to find a good "derivative" to a non-continuous function, we show how we can use a good smooth approximation and use its derivative in order to train. This is done by the simple observing that if we have stochastic (independent) weights \( w_{ij} \) then the activations \( y_i = \sum_j w_{ij} x_i \) are approximately Gaussian from the (Lyapunov) central limit theorem. This allows us to model the activation using a smooth distribution and use the reparametrization trick [11] to compute derivatives. The idea of modeling the distribution of activations instead of the distribution of weights was used in [10] for Gaussian weight distributions where it was called the local reparametrization trick. We show here that with small modifications it can be used to train discrete weights and not just continuous Gaussian distributions.

Figure 1: The top histogram in each subfigure shows the activation of a random neuron, \( y_i \), which is calculated in a straightforward feedforward setting when explicitly drawing the weights. The bottom shows samples from the approximated activation using Lyapunov CLT. Subfigure (a) refers to the first hidden layer whereas subfigure (b) refers to the last. We can see the approximation is very close to the actual activations when drawing weights. In addition, we see it even holds for the first hidden layer, where the number of elements is not large (in this example, 27 elements for a 3 \times 3 \times 3 convolution).

To conclude, in this paper we present a novel simple method for training neural networks with discrete weights. We show experimentally that we can train binary and ternary networks to achieve state-of-the-art results on most tasks, including ImageNet with the ResNet-18 network, compared to previously
proposed binary or ternary training algorithms.

2 Related Work

The closest work conceptually to ours is expectation-backpropagation \[17\] where the authors used the CLT approximation to train a Bayesian mean-field posterior. On a practical level, however, our training algorithm is significantly different and we show results on much larger networks than their work. Another close line of work is using the reparameterization trick for discrete variables using continuous approximations such as the Gumbal-softmax \[15 8\]. However since we are looking at the activation distribution, we can approximate using the simpler Gaussian distribution as seen in Fig 1.

The current leading works on training binary or ternary networks, \[1, 16, 13, 6\], take a different approach. They discretize during the forward pass, and backpropagate through this non-continuous discretization using the ”straight-through” estimator for the gradient. We note that \[16, 6, 17\] binarize the activations as well.

An alternative approach for reducing the memory and to speed-up deep networks is by weight compression \[14, 4\] that showed that deep networks can be highly compressed without significant loss of precision. The speedup capabilities of binary or ternary weight networks, especially on dedicated hardware, is however much higher.

3 Our method

We will now describe in detail our algorithm for training LR-nets, neural networks with discrete weights. We use a stochastic network model where each weight $W_{ij}$ is sampled independently from a multinomial distribution $W_{ij}$. Our goal is to minimize

$$
L(W) = E_{W \sim W} \left[ \sum_{i=1}^{N} \ell(h(x_i, W), y_i) \right] 
$$

3.1 Background

The standard approach for minimizing $L(W)$ with discrete distributions is by using the log-derivative trick

$$
\nabla L(W) = E_{W \sim W} \left[ \sum_{i=1}^{N} \ell(h(x_i, W), y_i) \nabla \log(P(W)) \right] 
$$

While this allows us to get an unbiased estimation of the gradient, it suffers from high variance which limits the effectiveness of this method.
For continuous distributions [11] suggested the reparameterization trick - instead of optimizing $E_{p(x)}[f(x)]$ for $p(x)$ we parametrize $x = g(\epsilon; \theta)$ where $\epsilon$ is drawn from a known fixed distribution $p(\epsilon)$ (usually Gaussian) and optimize $E_{p(\epsilon)}[f(g(\epsilon, \theta))]$ for $\theta$. We can sample $\epsilon_1, ..., \epsilon_m$ and use the Monte-Carlo approximation

$$\nabla \theta E_{p(\epsilon)}[f(g(\epsilon, \theta))] \approx \sum_{i=1}^{m} \nabla \theta f(g(\epsilon_i, \theta))$$  \hspace{1cm} (3)$$

In further work [10] it was noticed that if we are trying to learn Bayesian networks, sampling weights and running the model with different weights is quiet inefficient on GPUs. They observed that if the weight matrix $W$ is sampled from independent Gaussians $W_{ij} \sim N(\mu_{ij}, \sigma_{ij}^2)$ then the activations $y = Wx$ are distributed according to

$$y_i \sim N(\sum_j \mu_{ij}x_j, \sum_j \sigma_{ij}^2x_j^2)$$  \hspace{1cm} (4)$$

This allows you to sample activations instead of weights, which they called the local reparameterization trick, and it reduces run time considerably.

### 3.2 Discrete local reparameterization

The work in [10] focused on Gaussian weight distributions, but we will show how the local reparameterization allows us to optimize networks with discrete weights as well. Our main observation is that while eq. [4] holds true for Gaussians, from the (Lyapunov) central limit theorem we get that $y_i = \sum_{j} w_{ij}x_j$ should still be approximated well by the same Gaussian distribution $y_i \sim N(\sum_j \mu_{ij}x_j, \sum_j \sigma_{ij}^2x_j^2)$. The main difference is that $\mu_{ij}$ and $\sigma_{ij}^2$ are the mean and variance of a multinomial distribution not a Gaussian one. Once the discrete distribution has been approximated by a smooth one, we can compute the gradient of this smooth approximation and update accordingly.

This leads to a simple algorithm for training networks with discrete weights - let $\theta_{ij}$ be the parameters of the multinomial distribution over $w_{ij}$. At the forward pass given input vector $x$ we first compute the weights means $\mu_{ij} = E_{\theta_{ij}}[w_{ij}]$ and variances $\sigma_{ij}^2 = Var_{\theta_{ij}}[w_{ij}]$. We then compute the mean and variance of $y_i$, $m_i = E[y_i] = \sum_{j} \mu_{ij}x_j$ and $v_i^2 = Var[y_i] = \sum_{j} \sigma_{ij}^2x_j^2$. Finally we sample $\epsilon \sim N(0, I)$ and return $\hat{y} = m + v \odot \epsilon$, where $\odot$ stands for the element-wise multiplication.

During the backwards phase, given $\frac{\partial L}{\partial y}$ we can compute

$$\frac{\partial L}{\partial \theta_{ij}} = \frac{\partial L}{\partial y_i} \cdot \frac{\partial y_i}{\partial \theta_{ij}} = \frac{\partial L}{\partial y_i} \left( \frac{\partial m_{ij}}{\partial \theta_{ij}} + \epsilon_j \frac{\partial v_j}{\partial \theta_{ij}} \right) = \frac{\partial L}{\partial y_i} \left( x_i \frac{\partial \mu_{ij}}{\partial \theta_{ij}} + \frac{\epsilon_j x_j^2}{2v_j} \frac{\partial \sigma_{ij}^2}{\partial \theta_{ij}} \right)$$  \hspace{1cm} (5)$$

and similarly compute $\frac{\partial L}{\partial x_i}$ to backpropagate. We can then optimize using any first order optimization method, in our experiments we used Adam [9].
Algorithm 1 Forward pass

INPUT: Vector $x \in \mathbb{R}^{d \times n}$
PARAMETERS: Multinomial parameters $\theta_{ij}$ for each weight.

1: Compute $\mu_{ij} = \mathbb{E}_{\theta_{ij}}[w_{ij}]$ and $\sigma^2_{ij} = \text{Var}_{\theta_{ij}}[w_{ij}]$
2: Compute $m_i = \sum_j \mu_{ij} x_j$ and $v_i^2 = \sum_j \sigma^2_{ij} x_j^2$
3: Sample $\epsilon \sim \mathcal{N}(0, I)$
RETURN: $m + v \odot \epsilon$

4 Implementation Details

In section 3 we presented our main algorithmic approach, in this section we discuss the finer implementation details needed to achieve state of the art performance. We present the numerical experiments in section 5.

4.1 Initialization

In a regular DNN setting, we usually either initialize the weights with a random initializer, e.g xavier initializer [3] or set the initial values of the weights to some pretrained value from a classification network or another similar task. Here we are interested in initializing distributions over discrete weights from pretrained continuous deterministic weights, which we shall denote as $\tilde{W}$. We normalized $\tilde{W}$ beforehand to be mostly in the $[-1, 1]$ range, by dividing the weights in each layer by $\sigma_l$, the standard deviation of the weights in layer $l$.

We first explain our initialization in the ternary setting. Our aim is threefold: First, we wish to have our mean value as close as possible to the original full precision weight. Second, we wish to have low variance. Last, we do not want our initial distributions to be too deterministic, in order not to start at a bad local minima.

We initialize the probability of $p(w^l_{i,j} = 0)$ as follows:

$$p(w^l_{i,j} = 0) = p_{\text{max}} - (p_{\text{max}} - p_{\text{min}}) \cdot |\tilde{w}^l_{i,j}| \quad (6)$$

When $p_{\text{min}}$ and $p_{\text{max}}$ are hyperparameters (set to 0.05 and 0.95 respectively in our experiments). Using the initialization proposed in eq. 6 $p(w^l_{i,j} = 0)$ gets the maximum value $p_{\text{max}}$ when $\tilde{w}^l_{i,j} = 0$, and decays linearly to $p_{\text{min}}$ when $|\tilde{w}^l_{i,j}| = 1$. Next, we set $p(w^l_{i,j} = 1 \mid w^l_{i,j} \neq 0)$ so that the initialized mean of the discrete weight is equal to the original full precision weight:

$$\mathbb{E}[w^l_{i,j}] = [2 \cdot p(w^l_{i,j} = 1 \mid w^l_{i,j} \neq 0) - 1] \cdot (1 - p(w^l_{i,j} = 0)) \quad (7)$$

In order for the expression in eq. 7 to be equal to $\tilde{w}^l_{i,j}$, we need to set:

$$p(w^l_{i,j} = 1 \mid w^l_{i,j} \neq 0) = 0.5 \cdot (1 + \frac{\tilde{w}^l_{i,j}}{1 - p(w^l_{i,j} = 0)}) \quad (8)$$
We note that since we limited the range of $p(w_{i,j})$ this can lead to values larger than one or negative, hence finally, we clip the value returned from equations\[ to the range $[p_{min}, p_{max}]$. For a binary setting, we simply set $p(w_{i,j} = 0) = 0$.

4.2 Increasing Entropy

During some of our experiments with ternary networks, many of the weight distributions converge to a deterministic value and this led to bad performance. This is unwanted for two main reasons: First, this could lead to vanishing gradients due to saturation. Second, due to the small number of effective random variables our main assumption, that the CLT approximation hold is broken. An example of such case can be seen in fig. 2a.

Figure 2: Subfigure 2a is an example of a neuron from the first hidden layer at the end of the training, without a probability decay term. Since weight distributions converged to deterministic values, randomness is very small, causing the CLT approximation to break. Subfigure 2b is an example of a neuron from the same layer at the end of the training, but from an experiment with probability decay. Here our approximation holds much better.

This can be solved easily by adding $L_2$ regularization on the distribution weights (before sigmoid/softmax) to penalize very low entropy distributions. We emphasize that this regularization term was done mainly to help during training, not to reduce over-fitting. From here on we shall denote this regularization hyper-parameter as probability decay. We note that in practice only an extremely small probability decay is needed, but it had a surprisingly large effect on the overall performance. An example of such case can be seen in fig. 2b.

4.3 Reducing Entropy

In the binary setting we experienced the opposite. Since we do not have a weight for the middle value around 0, we found that many of the weight probabilities
learned are stuck in mid-values around 0.5 so that the expected value of that weight would remain around 0. That is not a wanted property, since we aim to get probabilities closer to 0 or 1 when we draw the final weights for evaluation. For this case, we add a beta density regularizer on our probabilities \( R(p) = p^{\alpha-1}(1 - p)^{\beta-1} \) for \( \alpha, \beta = 2 \). In general we note that unlike ternary weights, binary weights require more careful fine-tuning in order to perform well.

4.4 Testing

For evaluation, we would like to examine the DNN’s performance in the discrete setting. We draw the discrete weights from the learned distribution several times, and pick the weights that performed best on the validation set. We note that since the learned probabilities have low entropy, the variance between samples is quite low and this only gives a minor boost to the overall performance.

5 Experiments

We conduct extensive experiments on the MNIST, CIFAR-10 and ImageNet (ILSVRC2012) datasets. To show the flexibility of our method, we present results in both binary setting, in which weights are constrained +1 or -1, and ternary setting, in which weights are constrained to +1, 0 or -1. For the binary setting we compare our results with BinaryConnect [1], BNN [6], BWN and XNOR-Net [16]. For the ternary setting we compare our results with ternary weight networks [13].

In this section we show that in both experiment settings we obtain state-of-the-art results. Furthermore, while in previous work with binary networks, the first layer weights were in full precision, in our experiments the first layer is binary (or ternary) as well (with an exception of the binary ResNet-18), leading to an even larger memory and energy savings.

5.1 MNIST

MNIST is an image classification benchmark dataset, containing 60K training images and 10K test images from 10 classes of digits 0 – 9. Images are 28 \( \times \) 28 in gray-scale. For this dataset, we do not use any sort of preprocessing or augmentation. Following [13], we use the following architecture:

\[(32C5) - MP2 - (64C5) - MP2 - 512FC - Softmax\]  (9)

Where \( C5 \) is a 5 \( \times \) 5 ReLU convolution layer, \( MP2 \) is a 2 \( \times \) 2 max-pooling layer, and \( FC \) is a fully connected layer. We adopt Batch Normalization [7] into the architecture after every convolution layer. The loss is minimized with ADAM [9]. We use dropout [15] with a drop rate of 0.5. Weight decay parameter (on the last full precision layer) is set to \( 1e^{-4} \). We use a batch size of 256, initial learning rate is 0.01 and is divided by 10 after 100 epochs of training. For the binary setting, beta parameter is set to \( 1e^{-6} \). For the ternary setting,
probability decay parameter is set to $1e^{-11}$. We report the test error rate after 190 training epochs. The results are presented in Table 1.

5.2 CIFAR-10

CIFAR-10 is an image classification benchmark dataset \[12\], containing 50K training images and 10K test images from 10 classes. Images are $32 \times 32$ in RGB space. Each image is preprocessed by subtracting its mean and dividing by its standard deviation. During training, we pad each side of the image with 4 pixels, and a random $32 \times 32$ crop is sampled from the padded image. Images are randomly flipped horizontally. At test time, we evaluate the single original $32 \times 32$ image without any padding or multiple cropping. As in \[13\], we use the VGG inspired architecture:

\[
(2 \times 128C3) - MP2 - (2 \times 256C3) - MP2 - (2 \times 512C3) - MP2 - 1024FC - \text{Softmax}
\]

Where $C3$ is a $3 \times 3$ ReLU convolution layer, $MP2$ is a $2 \times 2$ max-pooling layer, and $FC$ is a fully connected layer. It is worth noting that \[1\] and \[6\] use a very similar architecture with two differences, namely adopting an extra fully connected layer and using an L2-SVM output layer, instead of softmax as in our experiments. We adopt Batch Normalization into the architecture after every convolution layer. The loss is minimized with ADAM \[9\]. We use dropout \[18\] with a drop rate of 0.5. Weight decay parameter is set to $1e^{-4}$. We use a batch size of 256, initial learning rate is 0.01 and is divided by 10 after 170 epochs of training. For the binary setting, beta parameter is set to $1e^{-6}$. For the ternary setting, probability decay parameter is set to $1e^{-11}$. We report the test error rate after 290 training epochs. The results are presented in Table 1.

5.3 ImageNet

ImageNet 2012 (ILSVRC2012) is a large scale image classification dataset \[2\], consisting of 1.28 million training images, and 50K validation images from 1000 classes. We adopt the proposed ResNet-18 \[5\], as in \[16\] and \[13\]. Each image is preprocessed by subtracting the mean pixel of the whole training set and dividing by the standard deviation. Images are resized so that the shorter side is set to 256. During training, the network is fed with a random $224 \times 224$ crop and images are randomly flipped horizontally. At test time, the center $224 \times 224$ crop is taken. We evaluate only with the single center crop. The loss is minimized with ADAM \[9\]. Weight decay parameter is set to $1e^{-5}$ and probability decay parameter is set to $1e^{-12}$. We use a batch size of 256, initial learning rate is 0.01 and is divided by 10 after 30 and 44 epochs of training. We report the test error rate after 55 training epochs. The results are presented in Table 2.
Table 1: Validation error rates on MNIST and CIFAR10 datasets, in a binary and ternary setting

|                      | MNIST       | CIFAR10     |
|----------------------|-------------|-------------|
| BinaryConnect        | 1.29 ± 0.08%| 9.9%        |
| BNN                  | 1.4%        | 10.15%      |
| Binary Weight Network | 0.95%      | 9.82%       |
| LR-net, binary (Ours)| **0.5%**   | **6.8%**    |
| Ternary Weight Network | 0.65%     | 7.44%       |
| LR-net, ternary (Ours)| **0.5%**  | **6.7%**    |
| Full precision reference | 0.5%     | 6.6%        |

Table 2: Validation error rates on ImageNet, in a binary and ternary setting

|                      | ImageNet (top-5) | ImageNet (top-1) |
|----------------------|------------------|------------------|
| Binary-Weight-Network| **17%**          | **39.2%**        |
| XNOR-Net             | 26.8%            | 48.8%            |
| LR-net, binary (Ours)| 18.8%            | 41.8%            |
| Ternary Weight Network | 15.8%          | 38.2%            |
| LR-net, ternary (Ours)| **15.2%**      | **36.5%**        |
| Full precision reference | 10.76%     | 30.43%           |

6 Conclusion

In this work we presented a simple, flexible and successful algorithm to train neural networks with discrete weights. To test our method and give evidence to its flexibility we trained and tested in binary and ternary settings, both in which a significant savings in memory and power consumption can be made. We presented results on various image classification datasets and reached state-of-the-are results in both settings on most datasets, paving the way to easier and more efficient training and inference of efficient low-power neural networks.

We note that the binary results on MNIST and CIFAR10 are a bit misleading - while they are very similar to the ternary network results, we needed quiet
a bit of work to achieve this, unlike ternary networks that worked well from
the start and on harder tasks they indeed performed better. We believe that
ternary networks are a much more reasonable model as they don’t force each
neuron to effect all next layer neurons, and allow us to model low-mean low-
variance stochastic weights. Further work into sparse ternary networks might
help reduce the extra runtime and memory compare to binary networks.

References

[1] Courbariaux, M., Bengio, Y., and David, J. Binaryconnect: Training
deep neural networks with binary weights during propagations. In Advances
in Neural Information Processing Systems, NIPS (2015).

[2] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In CVPR09
(2009).

[3] Glorot, X., and Bengio, Y. Understanding the difficulty of training
deep feedforward neural networks. In Proceedings of the Thirteenth Inter-
national Conference on Artificial Intelligence and Statistics (2010).

[4] Han, S., Mao, H., and Dally, W. J. Deep compression: Compress-
ing deep neural networks with pruning, trained quantization and huffman
coding.

[5] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for
image recognition. In Conference on Computer Vision and Pattern Recogni-
tion, CVPR (2016).

[6] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks. In Advances in Neural Information
Processing Systems (NIPS). 2016.

[7] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep
network training by reducing internal covariate shift. In ICML (2015).

[8] Jang, E., Gu, S., and Poole, B. Categorical reparameterization with
gumbel-softmax. In The International Conference on Learning Representa-
tions (ICLR) (2017).

[9] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimiza-
tion. In The International Conference on Learning Representations (ICLR)
(2015).

[10] Kingma, D. P., Salimans, T., and Welling, M. Variational dropout
and the local reparameterization trick. In Advances in Neural Information
Processing Systems (NIPS) (2015).
[11] Kingma, D. P., and Welling, M. Auto-encoding variational bayes. In *The International Conference on Learning Representations (ICLR)* (2014).

[12] Krizhevsky, A. Learning multiple layers of features from tiny images. In *Tech Report* (2009).

[13] Li, F., Zhang, B., and Liu, B. Ternary weight networks. *Advances in Neural Information Processing Systems (NIPS) Workshop* (2016).

[14] Louizos, C., Ullrich, K., and Welling, M. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems, NIPS, year = 2014*.

[15] Maddison, C. J., Mnih, A., and Teh, Y. W. The concrete distribution: A continuous relaxation of discrete random variables.

[16] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. *European Conference on Computer Vision, ECCV* (2016).

[17] Soudry, D., Hubara, I., and Meir, R. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *Advances in Neural Information Processing Systems, NIPS, year = 2014*.

[18] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15* (2014), 1929–1958.