# Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks

**Tim Salimans**                                                    SALIMANSTIM@GMAIL.COM
Aidence

**Diederik P. Kingma**                                              D.P.KINGMA@UVA.NL
University of Amsterdam, OpenAI

## Abstract

We present *weight normalization*: a reparameterization of the weight vectors in a neural network that decouples the length of those weight vectors from their direction. By reparameterizing the weights in this way we improve the conditioning of the optimization problem and we speed up convergence of stochastic gradient descent. Our reparameterization is inspired by batch normalization but does not introduce any dependencies between the examples in a mini-batch. This means that our method can also be applied successfully to recurrent models such as LSTMs and to noise-sensitive applications such as deep reinforcement learning or generative models, for which batch normalization is less well suited. Although our method is much simpler, it still provides much of the speed-up of full batch normalization. In addition, the computational overhead of our method is lower, permitting more optimization steps to be taken in the same amount of time. We demonstrate the usefulness of our method on applications in supervised image recognition, generative modelling, and deep reinforcement learning.

## 1. Introduction

*Deep learning* (Goodfellow et al., 2016) is a subfield of machine learning that consists in learning models that are wholly or partially specified by a class of flexible differentiable functions called neural networks. Learning usually entails the minimization or maximization of a scalar objective function, given the model and data, w.r.t. its parameters. In the case of deep neural networks, the set of parameters is composed of a large number of weights and biases. Optimization is typically performed by computing first-order gradients of the parameters w.r.t. the objective, and feeding them into a gradient-based optimization algorithm, iteratively performing small parameter updates from some initial point to a local optimum of the objective.

It is well known that the practical success of first-order gradient based optimization is highly dependent on the curvature of the objective that is optimized. If the condition number of the Hessian matrix of the objective at the optimum is low, the problem is said to exhibit *pathological curvature*, and first-order gradient descent will have trouble making progress (Martens, 2010; Sutskever et al., 2013). The amount of curvature, and thus the success of our optimization, is not invariant to reparameterization (Amari, 1997): There may be multiple equivalent ways of parameterizing the same model, some of which are much easier to optimize than others. Finding good ways of parameterizing neural networks is thus an important problem in deep learning.

While the architectures of neural networks differ widely across applications, they are typically mostly composed of conceptually simple computational building blocks sometimes called neurons: each such neuron computes a *weighted sum* over its real-valued inputs and adds a *bias* term, followed by the application of an elementwise nonlinear transformation. The weights and biases can be shared across neurons, and are (part of) the parameters of the model to be optimized over. Improving the general optimizability of deep networks is a challenging task (Glorot & Bengio, 2010), but since many neural architectures share these basic building blocks, improvements to the building blocks can improve the performance of a very wide range of model architectures.

In this paper, we propose a simple but general method, called *weight normalization*, for improving the optimizability of the weights of neural network models. The

method is inspired by *batch normalization* (Ioffe & Szegedy, 2015), another recently proposed reparameterization, but it is a deterministic method, and does not share batch normalization's property of increasing the stochasticity of the gradients.

We present *weight normalization*, our main contribution, in section 2. We then propose a data-dependent initialization strategy for the model parameters in section 3. In section 4 we discuss a hybrid between weight normalization and batch normalization, where we normalize the minibatch mean but not the variance, a method we call *mean-only batch normalization*. We empirically demonstrate the usefulness of our methods in section 5, with experiments in supervised image recognition, generative modelling, and deep reinforcement learning. In section 6 we discuss two possible extensions to our method that did not make it into our final experiments, and we discuss some of the properties of weight normalization. Finally, section 7 concludes.

## 2. Weight Normalization

We consider standard artificial neural networks where the computation of each neuron consists in taking a weighted sum of input features, followed by an elementwise nonlinearity:

$$y = \phi(\mathbf{w} \cdot \mathbf{x} + b), \tag{1}$$

where $\mathbf{w}$ is a $k$-dimensional weight vector, $b$ is a scalar bias term, $\mathbf{x}$ is a $k$-dimensional vector of input features, $\phi(.)$ denotes an elementwise nonlinearity such as the rectifier $\max(., 0)$, and $y$ denotes the scalar output of the neuron.

After associating a loss function to one or more neuron outputs, such a neural network is commonly trained by stochastic gradient descent in the parameters $\mathbf{w}, b$ of each neuron. In an effort to speed up the convergence of this optimization procedure, we propose to reparameterize each weight vector $\mathbf{w}$ in terms of a parameter vector $\mathbf{v}$ and a scalar parameter $g$ and to perform stochastic gradient descent with respect to those parameters instead. We do so by expressing the weight vectors in terms of the new parameters using

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v} \tag{2}$$

where $\mathbf{v}$ is a $k$-dimensional vector, $g$ is a scalar, and $||\mathbf{v}||$ denotes the Euclidean norm of $\mathbf{v}$. This reparameterization has the effect of normalizing the norm of the weight vector: we now have $||\mathbf{w}|| = g$, independent of the parameters $\mathbf{v}$. We therefore call this reparameterizaton *weight normalization*. By decoupling the norm of the weight vector ($g$) from the direction of the weight vector ($\mathbf{v}/||\mathbf{v}||$), we speed up convergence of our stochastic gradient descent optimization, as we show experimentally in section 5.

### 2.1. Gradients

Training a neural network in the new parameterization is done using standard stochastic gradient descent methods. Here we differentiate through (2) to obtain the gradient of a loss function $L$ with respect to the new parameters $\mathbf{v}, g$. Doing so gives

$$\begin{aligned} \nabla_g L &= \frac{\nabla_{\mathbf{w}} L \cdot \mathbf{v}}{||\mathbf{v}||} \\ \nabla_{\mathbf{v}} L &= \frac{g}{||\mathbf{v}||}\nabla_{\mathbf{w}} L - \frac{g \nabla_g L}{||\mathbf{v}||^2}\mathbf{v}, \end{aligned} \tag{3}$$

where $\nabla_{\mathbf{w}} L$ is the gradient with respect to the weights $\mathbf{w}$ as used normally. Backpropagation using weight normalization thus only requires a minor modification on top of standard neural network software. The expressions above are independent of the minibatch size and cause only minimal computational overhead.

### 2.2. Relation to batch normalization

An important source of inspiration for this reparameterization is *batch normalization* (Ioffe & Szegedy, 2015), which is equivalent to our method in the special case where our network only has a single layer, and the input features for that layer are whitened (independently distributed with zero mean and unit variance). Although exact equivalence does not usually hold for deeper architectures, we still find that our weight normalization method provides much of the speed-up of full batch normalization, while enjoying several advantages: The computational overhead of weight normalization is much lower, and it does not add stochasticity through the noisy estimates of minibatch statistics. This means that our method can also be applied to models like RNNs and LSTMs, as well as applications like Reinforcement Learning, for which batch normalization is less well suited.

## 3. Data-Dependent Initialization of Parameters

Besides a reparameterization effect, batch normalization also has the benefit of fixing the scale of the features generated by each layer of the neural network. This makes the optimization robust against parameter initializations for which these scales vary across layers. Since weight normalization lacks this property, we find it is important to properly initialize our parameters. We propose to sample the elements of $\mathbf{v}$ from a simple distribution with a fixed scale, which is in our experiments a normal distribution with mean zero and standard deviation 0.05. Before starting training, we then initialize the $b$ and $g$ parameters by performing an initial feedforward pass through our network with a minibatch of data, using the following computation

at each neuron:

$$
\begin{aligned}
t &= \frac{\mathbf{v} \cdot \mathbf{x}}{||\mathbf{v}||}, \\
y &= \phi\left(\frac{t - \mu[t]}{\sigma[t]}\right),
\end{aligned}
\tag{4}
$$

where $\mu[t]$ and $\sigma[t]$ are the mean and standard deviation of the pre-activation $t$ over the examples in the minibatch. We can then initialize bias $b$ and scale $g$ as

$$
\begin{aligned}
g &\leftarrow \frac{1}{\sigma[t]} \\
b &\leftarrow \frac{-\mu[t]}{\sigma[t]},
\end{aligned}
\tag{5}
$$

so that $y = \phi(\mathbf{w} \cdot \mathbf{x} + b)$. Like batch normalization, this method ensures that all features initially have zero mean and unit variance before application of the nonlinearity. With our method this only holds for the minibatch we use for initialization, and subsequent minibatches may have slightly different statistics, but experimentally we find this initialization method to work well. The method can also be applied to networks without weight normalization, simply by doing stochastic gradient optimization on the parameters $\mathbf{w}$ directly, after initialization in terms of $\mathbf{v}$ and $g$: this is what we compare to in section 5. Independently from our work, this type of initialization was recently proposed by different authors (Mishkin & Matas, 2015; Krähenbühl et al., 2015) who found such data-based initialization to work well for use with the normal parameterization.

The downside of this initialization method is that it can only be applied in similar cases as where batch normalization is applicable. For models with recursion, such as RNNs and LSTMs, we will have to resort to standard initialization methods.

## 4. Mean-only Batch Normalization

Weight normalization, as introduced in section 2, makes the scale of neuron activations approximately independent of the parameters $\mathbf{v}$. Unlike with batch normalization, however, the means of the neuron activations still depend on $\mathbf{v}$. We therefore also explore the idea of combining weight normalization with a special version of batch normalization, which we call *mean-only batch normalization*: With this normalization method, we subtract out the minibatch means like with full batch normalization, but we do not divide by the minibatch standard deviations. That is, we compute neuron activations using

$$
\begin{aligned}
t &= \mathbf{w} \cdot \mathbf{x}, \\
\tilde{t} &= t - \mu[t] + b, \\
y &= \phi(\tilde{t})
\end{aligned}
\tag{6}
$$

where $\mathbf{w}$ is the weight vector, parameterized using weight normalization, and $\mu[t]$ is the minibatch mean of the pre-activation $t$. During training, we keep a running average of the minibatch mean which we substitute in for $\mu[t]$ at test time.

The gradient of the loss with respect to the pre-activation $t$ is calculated as

$$
\nabla_t L = \nabla_{\tilde{t}} L - \mu[\nabla_{\tilde{t}} L],
\tag{7}
$$

where $\mu[.]$ denotes once again the operation of taking the minibatch mean. Mean-only batch normalization thus has the effect of centering the gradients that are backpropagated. This is a comparatively cheap operation, and the computational overhead of mean-only batch normalization is thus lower than for full batch normalization. In addition, this method causes less noise during training, and the noise that is caused is more gentle as the law of large numbers ensures that $\mu[t]$ and $\mu[\nabla_{\tilde{t}}]$ are approximately normally distributed. Thus, the added noise has much lighter tails than the highly kurtotic noise caused by the minibatch estimate of the variance used in full batch normalization. As we show in section 5.1, this leads to improved accuracy at test time.

## 5. Experiments

We experimentally validate the usefulness of our method using four different models for varied applications in supervised image recognition, generative modelling, and deep reinforcement learning.

### 5.1. Supervised Classification: CIFAR-10

To test our reparameterization method for the application of supervised classification, we consider the CIFAR-10 data set of natural images (Krizhevsky & Hinton, 2009). The model we are using is based on the ConvPool-CNN-C architecture of (Springenberg et al., 2015), with some small modifications: we replace the first dropout layer by a layer that adds Gaussian noise, we expand the last hidden layer from 10 units to 192 units, and we use $2 \times 2$ max-pooling, rather than $3 \times 3$. The only hyperparameter that we actively optimized (the standard deviation of the Gaussian noise) was chosen to maximize the performance of the network on a holdout set of 10000 examples, using the standard parameterization (no weight normalization or batch normalization). A full description of the resulting architecture is given in table 5.1.

We train our network for CIFAR-10 using Adam (Kingma & Ba, 2014) for 200 epochs, with a fixed learning rate and momentum of 0.9 for the first 100 epochs. For the last 100 epochs we set the momentum to 0.5 and linearly decay the learning rate to zero. We use a minibatch size of 100.

| Layer type | # channels | $x, y$ dimension |
|---|---|---|
| raw RGB input | 3 | 32 |
| ZCA whitening | 3 | 32 |
| Gaussian noise $\sigma = 0.15$ | 3 | 32 |
| $3 \times 3$ conv leaky ReLU | 96 | 32 |
| $3 \times 3$ conv leaky ReLU | 96 | 32 |
| $3 \times 3$ conv leaky ReLU | 96 | 32 |
| $2 \times 2$ max pool, str. 2 | 96 | 16 |
| dropout with $p = 0.5$ | 96 | 16 |
| $3 \times 3$ conv leaky ReLU | 192 | 16 |
| $3 \times 3$ conv leaky ReLU | 192 | 16 |
| $3 \times 3$ conv leaky ReLU | 192 | 16 |
| $2 \times 2$ max pool, str. 2 | 192 | 8 |
| dropout with $p = 0.5$ | 192 | 8 |
| $3 \times 3$ conv leaky ReLU | 192 | 6 |
| $1 \times 1$ conv leaky ReLU | 192 | 6 |
| $1 \times 1$ conv leaky ReLU | 192 | 6 |
| global average pool | 192 | 1 |
| softmax output | 10 | 1 |

*Table 1.* Neural network architecture for CIFAR-10.
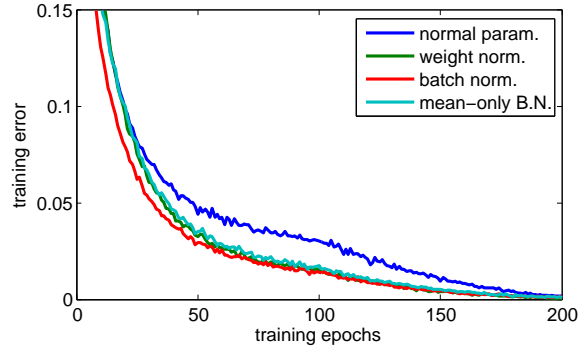


*Figure 1.* Training error for CIFAR-10 using different network parameterizations. For *weight normalization*, *batch normalization*, and *mean-only batch normalization* we show results using Adam with a learning rate of 0.003 for the first 100 epochs. For the normal parameterization this learning rate leads to divergence, so we instead use 0.0003 which gives the fastest optimization for this parameterization. For the last 100 epochs the learning rate is linearly decayed to zero.

We evaluate 4 different parameterizations of the network in table 5.1: 1) the standard parameterization, 2) using batch normalization, 3) using weight normalization, 4) using weight normalization combined with *mean-only* batch normalization. The network parameters are initialized using the scheme of section 3 such that all four cases have identical parameters starting out. For each case we pick the optimal learning rate in $\{0.0003, 0.001, 0.003, 0.01\}$. The resulting error curves during training can be found in figure 1: both weight normalization and batch normalization provide a significant speed-up over the standard parameterization. Batch normalization makes slightly more progress per epoch than weight normalization early on, although this is partly offset by the higher computational cost: with our implementation, training with batch normalization was about 16% slower compared to the standard parameterization. In contrast, weight normalization was not noticeably slower. During the later stage of training, weight normalization, batch normalization, and mean-only batch normalization all seem to optimize at about the same speed, with the normal parameterization still lagging behind.

After optimizing the network for 200 epochs using the different parameterizations, we evaluate their performance on the CIFAR-10 test set. The results are summarized in table 5.1: weight normalization and the normal parameterization have almost identical test accuracy ($\approx 8.45\%$ error). Batch normalization does significantly better at $8.05\%$ error: estimating the minibatch statistics adds noise during the training process, which has a beneficial regulariz-

| Model | Test Error |
|---|---|
| Maxout (Goodfellow et al., 2013) | 11.68% |
| Network in Network (Lin et al., 2014) | 10.41% |
| Deeply Supervised (Lee et al., 2014) | 9.6% |
| ConvPool-CNN-C (Springenberg et al., 2015) | 9.31% |
| ALL-CNN-C (Springenberg et al., 2015) | 9.08% |
| our CNN, weight normalization | 8.46% |
| our CNN, normal parameterization | 8.43% |
| our CNN, batch normalization | 8.05% |
| **our CNN, weight norm. + mean-only B.N.** | **7.31%** |

*Table 2.* Classification results on CIFAR-10 without data augmentation.

ing effect. Mean-only batch normalization (combined with weight normalization) has the best performance at $7.31\%$ test error. We hypothesize that the substantial improvement over regular batch normalization is due to the distribution of the noise caused by the normalization method during training: for mean-only batch normalization the minibatch mean has a distribution that is approximately Gaussian, while the noise added by full batch normalization during training has much higher kurtosis. As far as we are aware, the result with mean-only batch normalization represents the state-of-the-art for CIFAR-10 among methods that do not use data augmentation.
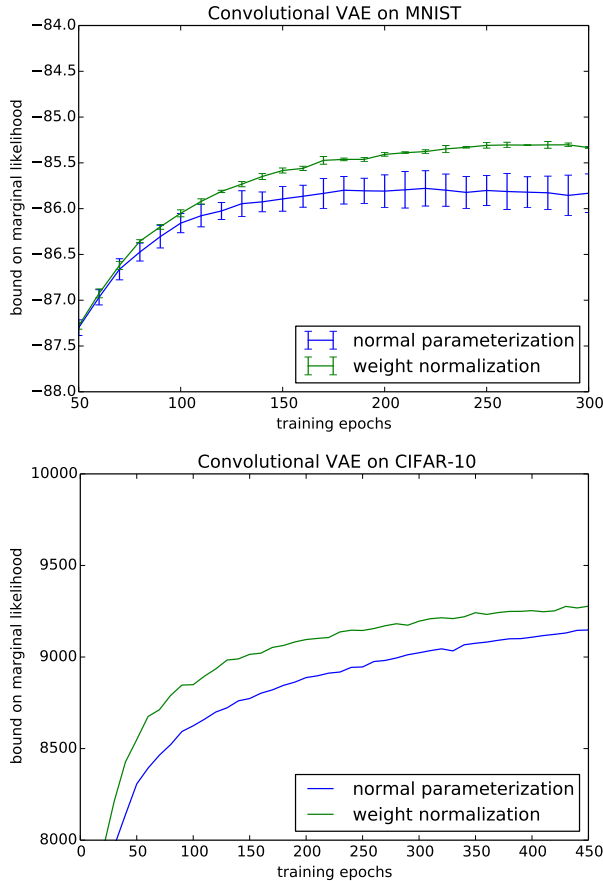
*Figure 2.* Marginal log likelihood lower bound on the MNIST (top) and CIFAR-10 (bottom) test sets for a convolutional VAE during training, for both the *standard* implementation as well as our modification with *weight normalization*. For MNIST, we provide standard error bars to indicate variance based on different initial random seeds.

## 5.2. Generative Modelling: Convolutional VAE

Next, we test the effect of weight normalization applied to deep convolutional variational auto-encoders (CVAEs) (Kingma & Welling, 2013; Rezende et al., 2014; Salimans et al., 2015), trained on the MNIST dataset of images of handwritten digits and the CIFAR-10 dataset of small natural images.

Variational auto-encoders are generative models that explain the data vector $\mathbf{x}$ as arising from a set of latent variables $\mathbf{z}$, through a joint distribution of the form $p(\mathbf{z}, \mathbf{x}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$, where the *decoder* $p(\mathbf{x}|\mathbf{z})$ is specified using a neural network. A lower bound on the log marginal likelihood $\log p(\mathbf{x})$ can be obtained by approximately inferring the latent variables $\mathbf{z}$ from the observed data $\mathbf{x}$ using an *encoder* distribution $q(\mathbf{z}|\mathbf{x})$ that is also specified as a neu-

ral network. This lower bound is then optimized to fit the model to the data.

We follow a similar implementation of the CVAE as in (Salimans et al., 2015) with some modifications, mainly that the encoder and decoder are parameterized with ResNet (He et al., 2015) blocks, and that the diagonal posterior is replaced with auto-regressive variational inference[1]. For MNIST, the encoder consists of 3 sequences of two ResNet blocks each, the first sequence acting on 16 feature maps, the others on 32 feature maps. The first two sequences are followed by a 2-times subsampling operation implemented using $2 \times 2$ stride, while the third sequence is followed by a fully connected layer with 450 units. The decoder has a similar architecture, but with reversed direction. For CIFAR-10, we used a neural architecture with ResNet units and multiple intermediate stochastic layers[1]. We used Adamax (Kingma & Ba, 2014) with $\alpha = 0.002$ for optimization, in combination with Polyak averaging (Polyak & Juditsky, 1992) in the form of an exponential moving average that averages parameters over approximately 10 epochs.

In figure 2, we plot the test-set lower bound as a function of number of training epochs, including error bars based on multiple different random seeds for initializing parameters. As can be seen, the parameterization with weight normalization has lower variance and converges to a better optimum. We observe similar results across different hyper-parameter settings.

## 5.3. Generative Modelling: DRAW

Next, we consider DRAW, a recurrent generative model by (Gregor et al., 2015). DRAW is a variational auto-encoder with generative model $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ and encoder $q(\mathbf{z}|\mathbf{x})$, similar to the model in section 5.2, but with both the encoder and decoder consisting of a recurrent neural network comprised of Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) units. LSTM units consist of a memory cell with additive dynamics, combined with input, forget, and output gates that determine which information flows in and out of the memory. The additive dynamics enables learning of long-range dependencies in the data.

At each time step of the model, DRAW uses the same set of weight vectors to update the *cell states* of the LSTM units in its encoder and decoder. Because of the recurrent nature of this process it is not clear how batch normalization could be applied to this model: Normalizing the cell states diminishes their ability to pass through information. Fortunately, weight normalization can be applied trivially to the weight vectors of each LSTM unit, and we find this to work well empirically.
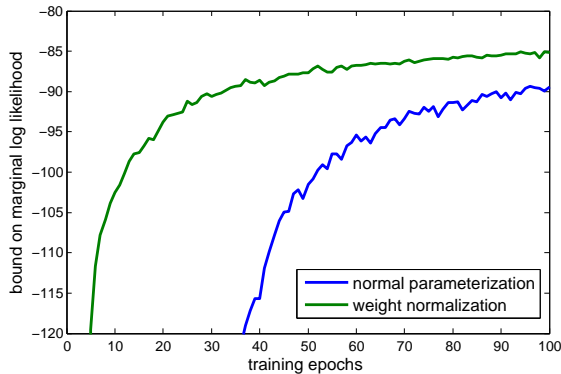
---

[1]Manuscript in preparation

*Figure 3.* Marginal log likelihood lower bound on the MNIST test set for DRAW during training, for both the *standard* implementation as well as our modification with *weight normalization*. 100 epochs is not sufficient for convergence for this model, but the implementation using weight normalization clearly makes progress much more quickly than with the standard parameterization.



*Figure 4.* Evaluation scores for Space Invaders obtained by DQN after each epoch of training, for both the standard parameterization and using weight normalization. Learning rates for both cases were selected to maximize the highest achieved test score.

We take the Theano implementation of DRAW provided at https://github.com/jbornschein/draw and use it to model the MNIST data set of handwritten digits. We then make a single modification to the model: we apply weight normalization to all weight vectors. As can be seen in figure 3, this significantly speeds up convergence of the optimization procedure, even without modifying the initialization method and learning rate that were tuned for use with the normal parameterization.

## 5.4. Reinforcement Learning: DQN

Next we apply weight normalization to the problem of Reinforcement Learning for playing games on the Atari Learning Environment (Bellemare et al., 2013). The approach we use is the Deep Q-Network (DQN) proposed by (Mnih et al., 2015). This is an application for which batch normalization is not well suited: the noise introduced by estimating the minibatch statistics destabilizes the learning process. We were not able to get batch normalization to work for DQN without using an impractically large minibatch size. In contrast, weight normalization is easy to apply in this context, as is the initialization method of section 3. Stochastic gradient learning is performed using Adamax (Kingma & Ba, 2014) with momentum of 0.5. We search for optimal learning rates in $\{0.0001, 0.0003, 0.001, 0.003\}$, generally finding 0.0003 to work well with weight normalization and 0.0001 to work well for the normal parameterization. We also use a larger minibatch size (64) which we found to be more efficient on our hardware (Amazon Elastic Compute Cloud g2.2xlarge GPU instance). Apart from these changes
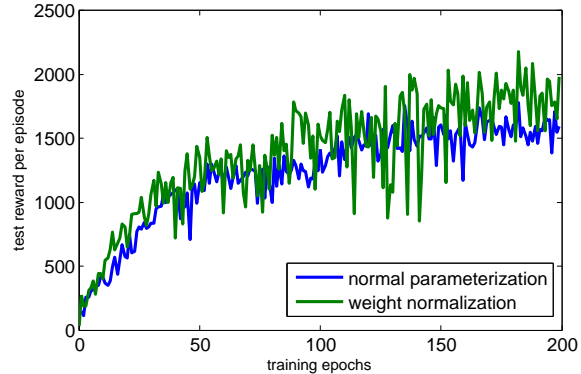
we follow (Mnih et al., 2015) as closely as possible in terms of parameter settings and evaluation methods. However, we use a Python/Theano/Lasagne reimplementation of their work, adapted from the implementation available at https://github.com/spragunr/deep_q_rl, so there may be small additional differences in implementation.

Figure 4 shows the training curves obtained using DQN with the standard parameterization and with weight normalization on Space Invaders. Using weight normalization the algorithm progresses more quickly and reaches a better final result. Table 5.4 shows the final evaluation scores obtained by DQN with weight normalization for four games: on average weight normalization improves the performance of DQN.

| Game | normal | weight norm. | Mnih et al. |
|---|---|---|---|
| Breakout | 410 | 403 | 401 |
| Enduro | 1,250 | 1,448 | 302 |
| Seaquest | 7,188 | 7,375 | 5,286 |
| Space Invaders | 1,779 | 2,179 | 1,975 |

*Table 3.* Maximum evaluation scores obtained by DQN, using either the normal parameterization or using weight normalization. The scores indicated by *Mnih et al.* are those reported by (Mnih et al., 2015): Our normal parameterization is approximately equivalent to their method. Differences in scores may be caused by small differences in our implementation.

# 6. Extensions

We investigated multiple extensions to the weight normalization method as presented above. Here we discuss two extensions that we did not end up using in our final experiments, but that may be interesting to investigate further in future work.

## 6.1. Exponential parameterization of the scale parameter $g$

An alternative parameterization of the weight-scale is $g = e^{cs}$, where $s$ is a log-scale parameter to learn by stochastic gradient descent, and $c$ is a constant that throttles the effective learning rate of $s$. Parameterizing the $g$ parameter in the log-scale is more natural, and more easily allows it to span a wide range of different magnitudes. We performed various experiments with this parameterization with various values of $c$. We found that a value of $c$ between 1 and 5 worked well, where a value of 3 seemed about optimal for our experiments. However, the eventual test-set performance was not significantly better or worse than the results with directly learning $g$ in its original parameterization, so we did not use the log-scale parameterization in our final experiments.

## 6.2. Fixing the norm of $\mathbf{v}$

When learning a neural network with weight normalization using standard gradient descent, the norm of $\mathbf{v}$ grows monotonically with the number of weight updates, which decreases the effective learning rate on the weights in our network. This effect arises because the unnormalized weight vector $\mathbf{v}$ only enters our model through its normalized version $\mathbf{v}/||\mathbf{v}||$, which ensures that the gradient $\nabla_{\mathbf{v}} L$ of any loss function $L$ is necessarily orthogonal to $\mathbf{v}$. That is, $\nabla_{\mathbf{v}} L \cdot \mathbf{v} = 0$, which can be verified by plugging in the expression for the gradient given in (3). Suppose now that an optimizer updates the weight vector $\mathbf{v}$ to $\mathbf{v}' = \mathbf{v} + \Delta\mathbf{v}$, where $\Delta\mathbf{v} \propto \nabla_{\mathbf{v}} L$ (steepest ascent/descent), and that the relative norm of the update w.r.t. the old weight vector is $||\Delta\mathbf{v}||/||\mathbf{v}|| = c$. The new weight vector then has norm $||\mathbf{v}'|| = \sqrt{||\mathbf{v}||^2 + c^2||\mathbf{v}||^2}$, as dictated by the Pythagorean theorem. Therefore, the norm of the unnormalized weights is increased by the factor $||\mathbf{v}'||/||\mathbf{v}|| = \sqrt{||\mathbf{v}||^2 + c^2||\mathbf{v}||^2}/||\mathbf{v}|| = \sqrt{1 + c^2} > 1$.

This property of monotonically increasing norm $||\mathbf{v}||$ does not strictly hold for optimizers that use separate learning rates for individual parameters, like Adam (Kingma & Ba, 2014) which we use in experiments, where $||\mathbf{v}||$ can either grow or shrink during optimization; still, we found that the norm would generally grow during optimization, resulting in decreasing effective learning rate. To eliminate this property, we experimented with imposing a hard constraint in the form of $||\mathbf{v}|| \leq a$, for some constant $a$, to prevent the unnormalized weights from growing overly large. However, for a reasonable initialization of $\mathbf{v}$ (e.g. as in section 3), we did not find that imposing this upper bound resulted in noticeable improvements of convergence speed.

Instead, we found that the ability to grow the norm $||\mathbf{v}||$ makes optimization of neural networks with weight normalization very robust to the value of the learning rate: If the learning rate is too large, the norm of the unnormalized weights grows quickly until an appropriate effective learning rate is reached. Once the norm of the weights has grown large with respect to the norm of the updates, the effective learning rate stabilizes as $\sqrt{1 + ||\Delta\mathbf{v}||^2/||\mathbf{v}||^2} \approx 1$ for $||\Delta\mathbf{v}|| \ll ||\mathbf{v}||$. Experimentally, we found that neural networks with weight normalization therefore work well with a much wider range of learning rates than when using the normal parameterization. It has been observed that neural networks with batch normalization also have this property (Ioffe & Szegedy, 2015), which can be explained with a similar analysis.

Although bounding the norm of the unnormalized weights was not found to speed up convergence, it can be used to obtain a regularizing effect during training: For a small choice of upper bound, e.g. $||\mathbf{v}|| \leq 0.25$, the parameter updates remain relatively large, which adds noise due to the random sampling of training examples. This can lead to good test-set performance when the parameters are averaged over many epochs (i.e. Polyak averaging, see (Polyak & Juditsky, 1992)). For variational auto-encoders (section 5.2) we found that this effect was often beneficial, but it was not helpful with our experiments on supervised classification and reinforcement learning. For consistency, all results reported in this paper are therefore without any upper bound on $||\mathbf{v}||$.

# 7. Conclusion

We have presented *weight normalization*, a simple reparameterization of the weight vectors in a neural network that accelerates the convergence of stochastic gradient descent optimization. Weight normalization was applied to four different models in supervised image recognition, generative modelling, and deep reinforcement learning, showing a consistent advantage across applications. The reparameterization method is easy to apply, has low computational overhead, and does not introduce dependencies between the examples in a minibatch, making it our default choice in the development of new deep learning architectures.

## References

Amari, S. Neural learning in structured parameter spaces - natural Riemannian gradient. In *Advances in Neural Information Processing Systems*, pp. 127–133. MIT Press, 1997.

Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Bergstra, James, Goodfellow, Ian J., Bergeron, Arnaud, Bouchard, Nicolas, and Bengio, Yoshua. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 06 2013.

Glorot, Xavier and Bengio, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron. Deep learning. Book in preparation for MIT Press, 2016.

Goodfellow, Ian J, Warde-Farley, David, Mirza, Mehdi, Courville, Aaron, and Bengio, Yoshua. Maxout networks. In *ICML*, 2013.

Gregor, Karol, Danihelka, Ivo, Graves, Alex, and Wierstra, Daan. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kingma, Diederik P and Welling, Max. Auto-Encoding Variational Bayes. *Proceedings of the 2nd International Conference on Learning Representations*, 2013.

Krähenbühl, Philipp, Doersch, Carl, Donahue, Jeff, and Darrell, Trevor. Data-dependent initializations of convolutional neural networks. *arXiv preprint arXiv:1511.06856*, 2015.

Krizhevsky, Alex and Hinton, Geoffrey. Learning multiple layers of features from tiny images, 2009.

Lee, Chen-Yu, Xie, Saining, Gallagher, Patrick, Zhang, Zhengyou, and Tu, Zhuowen. Deeply-supervised nets. In *Deep Learning and Representation Learning Workshop, NIPS*, 2014.

Lin, Min, Qiang, Chen, and Yan, Shuicheng. Network in network. In *ICLR: Conference Track*, 2014.

Martens, James. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 735–742, 2010.

Mishkin, Dmytro and Matas, Jiri. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Polyak, Boris T and Juditsky, Anatoli B. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.

Rezende, Danilo J, Mohamed, Shakir, and Wierstra, Daan. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 1278–1286, 2014.

Salimans, Tim, Kingma, Diederik P, and Welling, Max. Markov chain Monte Carlo and variational inference: Bridging the gap. In *ICML*, 2015.

Springenberg, Jost Tobias, Dosovitskiy, Alexey, Brox, Thomas, and Riedmiller, Martin. Striving for simplicity: The all convolutional net. In *ICLR Workshop Track*, 2015.

Sutskever, Ilya, Martens, James, Dahl, George, and Hinton, Geoffrey. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pp. 1139–1147, 2013.