UNIVERSITY OF CALIFORNIA

SANTA CRUZ

A study of the Exponentiated Gradient +/- algorithm for stochastic optimization of neural networks

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

David F. Parks

September 2019

The Thesis of David F. Parks
is approved:

Professor Manfred K. Warmuth

Professor Shawfeng Dong

Professor J. Xavier Prochaska

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

Contents

List of Figures **v**

List of Tables **vii**

Abstract **viii**

Acknowledgements **ix**

1 Introduction **1**
- 1.1 Stochastic Gradient Descent Optimization 1
- 1.2 Gradient descent algorithm variants 2
  - 1.2.1 Vanilla SGD 4
  - 1.2.2 SGD with Momentum 4
  - 1.2.3 Nesterov accelerated gradient descent 5
  - 1.2.4 Adagrad 8
  - 1.2.5 Adadelta 9
    - 1.2.5.1 Adadelta Initialization Issues 11
  - 1.2.6 Window Grad 12
  - 1.2.7 RMSprop 13
  - 1.2.8 Adam 14

2 Exponentiated Gradient $\pm$ Update Algorithm **20**
  - 2.0.1 Exponentiated Gradient 21
  - 2.0.2 Exponentiated Gradient $\pm$ 21
    - 2.0.2.1 Memory footprint and computation cost 23
    - 2.0.2.2 U Scaling 24
    - 2.0.2.3 Normalization method 26
  - 2.0.3 EG$^{\pm}$ update algorithm 29
  - 2.0.4 Concrete Implementations 30
    - 2.0.4.1 EG$^{\pm}$ code from ConvnetJS 32
    - 2.0.4.2 EG$^{\pm}$ Initialization 32
  - 2.0.5 Visualization 33

3 Findings and conclusions **35**
- 3.1 Summary of findings 35
- 3.2 U parameter scaling 36

3.3 Distribution of weights produced by EG+- vs. Nesterov in a CNN 38
3.4 Unnormalized EG 40
3.5 EG+/- with random noisy features 41
3.6 Comparing EG$^{\pm}$ with other optimization methods 42
3.7 Applying adadelta's per-weight learning rate to EG$^{\pm}$ 45
3.8 Sum of square loss vs. cross entropy 47
3.9 EG+- compared vs. SGD/L1 regularization 48
3.10 Sharing weights applied to EG$^{\pm}$ 48
3.11 Overfitting with SGD vs EG$^{\pm}$ 51
3.12 Results of applying EG+- in a convolutional neural network . . . 52
3.13 EG$^{\pm}$ on residual neural networks 55
3.14 Adversarial examples 56
3.15 Conclusion 58

Bibliography **60**

List of Figures

1.1 SGD vs. Momentum 6
1.2 Momentum vs. Nesterov 7
1.3 Nesterov vs. Adagrad 9
1.4 Adagrad vs. Adadelta 11
1.5 Adadelta Initialization 16
1.6 Adadelta vs. WidnowGrad 17
1.7 Adadelta vs. RMSProp 18
1.8 RMSProp vs. Adam 19

2.1 EG vs. SGD convergence 24
2.2 UScaling convergence 26
2.3 Visualization of normalization methods 27
2.4 RMSPRop vs. EG 34

3.1 Choosing a U parameter 37
3.2 U scaling parameter experiment 38
3.3 Weights distribution of EG$^{\pm}$ vs. Nesterov 39
3.4 Biases distribution of EG$^{\pm}$ vs. Nesterov 39
3.5 EG$^{\pm}$ Unnormalized 40
3.6 50% random noise experiment 41
3.7 10k features of random Gaussian noise experiment 42
3.8 EG vs. other optimizers 43

3.9 EG with adaptive learning rate 47
3.10 EG with L1 regularization experiment 49
3.11 Weight sharing applied to EG$^{\pm}$ 50
3.12 Weight sharing experiment detailed plot 51
3.13 Comparing EG$^{\pm}$ to SGD with L2 regularization 52
3.14 CNN results on CIFAR10 dataset 54
3.15 Activation values of EG$^{\pm}$ vs. Nesterov 54
3.16 Adversarial examples 57

List of Tables

3.1 Comparing trainers on MNIST, displaying test set accuracy over time. Rows are per number of samples trained on, columns are per optimization algorithm. 44
3.2 Comparing all gradient optimization methods on a 2 layer MNIST dataset, including EG$^{\pm}$ with adaptive learning rate from Adadelta, the adaptive EG$^{\pm}$ uses a global learning rate parameter of 0.02, U=40, and ρ=0.95. 46
3.3 EG vs other optimizers compared using a 32 layer residual network on the cifar10 dataset. 55
3.4 Comparison of EG optimizer with varying normalization method against other optimizers using 1000 test cifar10 samples and adversarial versions of the same images. 58

Abstract

A study of the Exponentiated Gradient +/- algorithm for stochastic optimization of neural networks

by David F. PARKS

Exponentiated Gradient +/- (abbr. EG$^{\pm}$) is a gradient update algorithm drawn from work by Manfred Warmuth (Kivinen and Warmuth, 1997) in the online learning setting. This thesis ports the algorithm into the context of deep neural networks and analyses its fitness in that context compared to the current state of the art gradient update methods. Existing methods employ an additive update scheme whereby some fraction of the gradient is added to the weight values to update them at each iteration in the gradient descent algorithm. EG$^{\pm}$ provides a multiplicative update scheme whereby a proportion of the gradient is multiplied into the original weight value, and then normalized to update the weight. EG$^{\pm}$ is motivated by using a relative entropy regularization. This thesis analyzes various properties and experimental results of the algorithm in comparison to other update methods, and analyzes EG$^{\pm}$ in the context of state of the art residual networks and challenging vision problems. Three published implementations are experimented with, and demonstrate that EG$^{\pm}$ performs better than SGD when there are many noisy features, and that it compares well with commonly used state-of-the art gradient descent optimization methods. EG$^{\pm}$ also performs better than most SGD based optimizers on black-box adversarial attacks, with the exception of non momentum based SGD with which it performs similarly.

Acknowledgements

Foremost, I would like to thank Professor Manfred Warmuth for overseeing this work and providing guidance, inspiration, and for introducing me to many talented experts in the field from across the globe. I also want especially mention Professor Shawfeng Dong who both mentored me in aspects of optimization theory as well as advised me on multiple projects involving implementations of neural networks on multiple frameworks, and in cluster computing with neural networks. The experience gained in my work with Professor Dong has benefited this work in innumerable ways. I would like to thank Professor J. Xavier Prochaska from Astronomy for supporting work I've done in applying neural networks to problems in the Astronomy domain, which has provided me with a much deeper intuition and understanding of the algorithms applied in this thesis. I would like to thank Professor Ram Akella, and Dr. Jay Pujara, whom I've learned from and worked on related research projects, all of whom have provided me tremendous amounts of information that went directly into this work and all of whom have helped with research projects that directly affected my knowledge in this topic area. I would like to thank Ryan Hausen, a fellow graduate student with whom I've often collaborated with and bounced uncountably many ideas off of since starting this work.

Chapter 1

Introduction

This thesis will introduce the Exponentiated Gradient $\pm$ (EG$^{\pm}$) algorithm in Chapter 2 as an alternative gradient descent optimization method. In Chapter 3 we will explore the properties of EG$^{\pm}$ and compare the algorithm to 8 of the most commonly used optimization algorithms used in neural network training today. The core conclusions we derive are that EG$^{\pm}$ performs or or near state of the art on common datasets such as MNIST and CIFAR10; it performs well on simple fully connected networks as well as the more complex architectures of residual networks (ResNet); it performs better with features consisting of random noise than modern optimizers; and EG$^{\pm}$ shares beneficial properties with vanilla SGD (stochastic gradient descent) on black box adversarial attacks that other optimizers perform poorly on.

Before introducing EG$^{\pm}$ let's take a tour through the existing, gradient descent optimizers, starting with vanilla SGD.

1.1 Stochastic Gradient Descent Optimization

Stochastic Gradient descent is the workhorse algorithm for training artificial neural networks today. Stochastic gradient descent (SGD) is the most basic method in a family of gradient descent update algorithms. SGD is a simple, efficient, and effective algorithm for updating the weights of the network given the gradient. The update rule for SGD is simply: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \nabla_{\boldsymbol{w}} J(\boldsymbol{w}_t)$, where $\boldsymbol{w}$ is a vector of weights (the networks trainable weights and biases for each layer, and other trainable weights such as γ and β used in batch normalization), t is the training iteration of the algorithm, and η is the learning rate, a (typically small) step size which is specified as a hyper-parameter of the algorithm, J is a loss function that evaluates the fitness of the network for a set of weights. Given the gradient of the loss function with respect to the weights SGD takes a fraction of the gradient and subtracts it from the weights, taking a linearly approximated step in the direction of the negative gradient.

A single update step in SGD can be applied using a gradient computed over (a) one sample of the data (stochastic gradient descent), (b) a mini-batch of uniformly random samples (mini-batch gradient descent), or (c) the entire data set (full batch gradient descent). For notational brevity this document always assumes updates are taken with respect to a mini-batch of samples, as is typical in practice.

A number of variants of SGD have been developed that provide improvements to the basic SGD algorithm. These algorithms add concepts such as momentum, per-weight learning rates, and other beneficial features. The family of commonly used gradient descent optimization algorithms is discussed in the following sections.

1.2 Gradient descent algorithm variants

We will start by reviewing the most popular optimization methods (Ruder, 2016; Pascanu et al., 2013) employed by today's neural network frameworks. For each of the update algorithms there will be a visualization of how the algorithm deals with a prototypical problem in optimization, which is exemplified by a plateauing in one dimension $x1$, and a steep gradient in another dimension x_2. Schaul et al. (2013) propose this among a number of unit tests to assess the suitability of an update algorithm.

The following gradient update algorithms are all implemented by the major frameworks such as Tensorflow, Torch, Caffe, etc.

- **SGD (Stochastic Gradient descent)** — The original and most basic form of gradient based optimization methods.
- **SGD with Momentum** — Adds a momentum term that alleviates the zigzag problem of SGD.
- **Nesterov** — Performs momentum updates more intelligently.
- **Adagrad** — Incorporates per-weight learning rate which decays over time.
- **Adadelta** — Ameliorates the decaying learning rate of Adagrad while maintaining per-weight learning rates.
- **Window Grad** — This method was published as "idea 1" with Adadelta and restricts the time period over which we accumulate the gradients to a window.
- **RMSProp** — Another variant on Adagrad, similar to Adadelta which computes a per-weight learning rate, with some benefits over Adadelta.
- **Adam** — Applies concepts of both momentum and adaptive weights while correcting for initialization bias.

Notable among these algorithms is that they all employ an additive update method just as SGD does. In SGD, $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \nabla_{\boldsymbol{w}} J(\boldsymbol{w}_t)$, a fraction of the negative gradient is added to the previous weights. To contrast that, a multiplicative update method multiplies each weight by an exponential factor that has the ith component of the negative gradient in the exponent. In practice this has the effect of being a relatively small number slightly above or below one which gets multiplied into the existing weight in the update process. The family of exponentiated gradient optimization algorithms fall under this second paradigm, which we we will introduce in Chapter 2, and study in Chapter 3.

Assuming the reader is familiar with stochastic gradient descent, let's look at each of SGD's variants in turn with a visualization of each algorithm optimizing a 2D prototypical example. Each of these algorithms have been re-implemented for this project in Matlab to produce these visualizations.

1.2.1 Vanilla SGD

In vanilla SGD, gradient descent updates are performed by:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \nabla_{\boldsymbol{w}} J(\boldsymbol{w}_t) \tag{1.1}$$

$\nabla_{\boldsymbol{w}} J$	the gradient of the weights with respect to the loss function. This is the gradient calculated by backprop.
η	is a scalar learning rate provided to the algorithm as a hyperparameter.
$\boldsymbol{w}$	is a vector of all network weights.
t	is a scalar time step, representing the iteration comprising a forward pass for making a prediction, and a backward pass to compute the gradients.

SGD performs its update by taking a linear step in the direction of the negative gradient and then recomputing the weights and gradient at that point.

1.2.2 SGD with Momentum

The first addition to vanilla SGD is adding a momentum term (Qian, 1999). The momentum term is akin to the speed gained by a ball rolling downhill. At the top of the hill the ball starts rolling slowly, building up momentum as it continues downhill. It will reach a maximum terminal velocity depending on the medium it's traveling through (air for example). This has a particularly beneficial effect when traveling down a valley (Sutton, 1986). When the gradient surface provides a path towards a better optimum in one direction, but a steep gradient in other directions SGD is known to oscillate, slowing progress towards the objective. The momentum term minimizes progress along the axis of oscillation, and increases momentum along the axis where the gradient doesn't change.

The SGD update with momentum is given by the equations:

$$\boldsymbol{v}_{t+1} = \gamma \boldsymbol{v}_t + \eta \nabla_{\boldsymbol{w}} J(\boldsymbol{w}_t) \tag{1.2}$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \boldsymbol{v}_{t+1} \tag{1.3}$$

$\nabla_{\boldsymbol{w}} J$	the gradient of the weights with respect to the loss function. This is the gradient calculated by backprop.
η	is a scalar learning rate provided to the algorithm as a hyperparameter.
$\boldsymbol{v}$	a vector of computed velocity terms per each weight $\boldsymbol{w}$.
$\boldsymbol{w}$	is a vector of all network weights.
t	is a scalar time step, representing the iteration comprising a forward pass for making a prediction, and a backward pass to compute the gradients.

In Figure 1.1 the oscillation effect of SGD (in red) is quite visible, whereas momentum draws a path that appears much more "natural".

1.2.3 Nesterov accelerated gradient descent

Nesterov accelerated gradient descent (Nesterov, 1983) makes a small, but beneficial change to the concept of momentum. Nesterov updates take the momentum of the previous update into account before computing the gradient, then upon computing the gradient takes a "correction" step. Whereas momentum

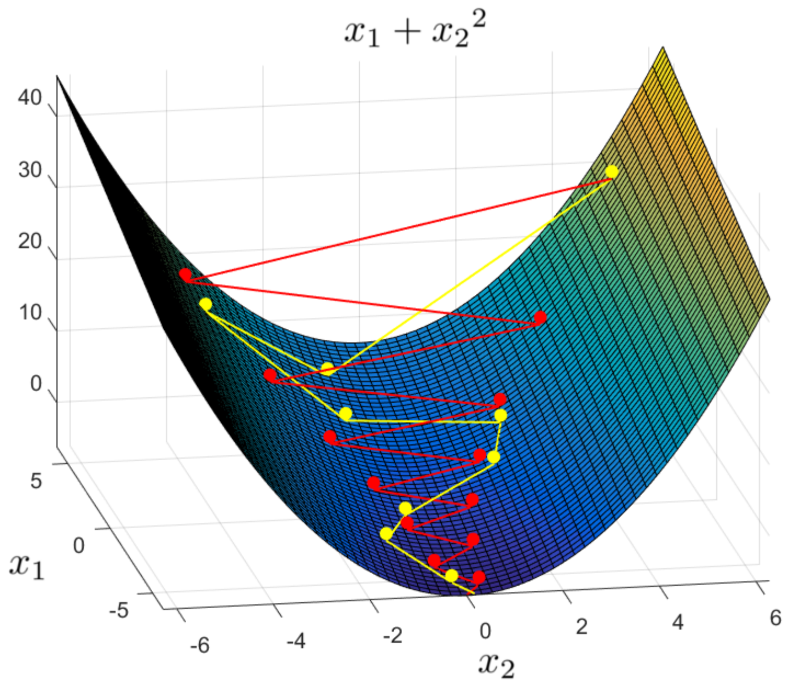


FIGURE 1.1: Momentum updates in yellow, vanilla SGD in red. SGD can oscillate back and forth in a valley, but momentum offers a smoothing effect that draws it out of the oscillation pattern.

simply takes a step in the gradient direction. This can be thought of essentially as a reordering of the optimization process where momentum is calculated first before taking a step and then intelligently correcting for any error that occurred.

On convex problems Nesterov's approach has provably better bounds than SGD with Momentum. However the theoretical guarantees are imperfect in the face of stochastic gradient noise due to mini batches of samples not containing a perfect gradient. This is discussed in more detail in Sutskever's thesis (Sutskever, 2013).

In the non convex setting, there are no theoretical guarantees about how Nesterov will perform, however this, update-then-correct strategy has been demonstrated to perform more stably in a wide variety of cases.

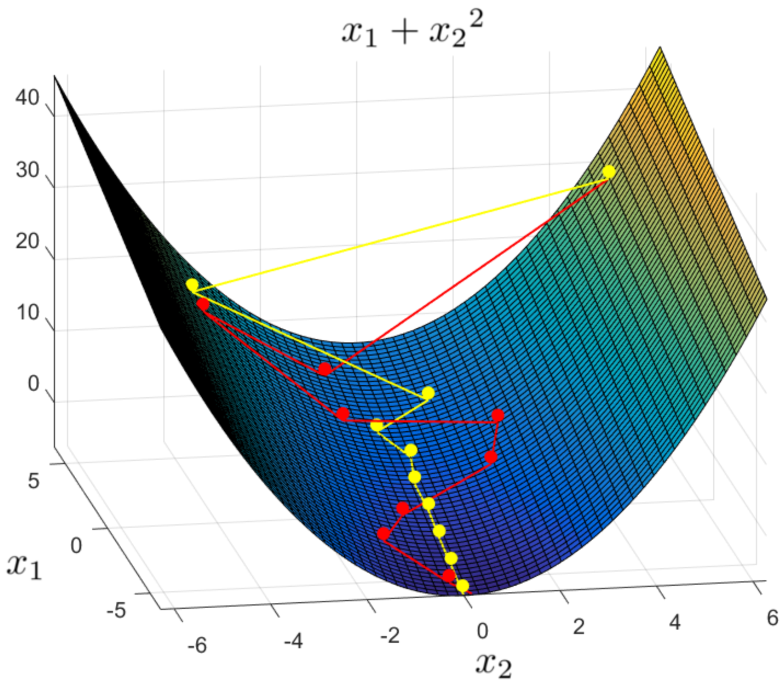


FIGURE 1.2: Momentum update in red, Nesterov update in yellow. Nesterov updates take momentum from the previous step into account prior to deciding where to compute the gradient, then computes the gradient and takes a "corrected" gradient step, smoothing out the progression and still avoiding SGD's oscillation.

The Nesterov accelerated gradient update is given by the equations:

$$\boldsymbol{v}_{t+1} = \gamma \boldsymbol{v}_t + \eta \nabla_{\boldsymbol{w}} J(\boldsymbol{w}_t - \gamma \boldsymbol{v}_t) \tag{1.4}$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \boldsymbol{v}_{t+1} \tag{1.5}$$

$\nabla_{\boldsymbol{w}} J$	the gradient of the weights with respect to the loss function. This is the gradient calculated by backprop.
η	is a scalar learning rate provided to the algorithm as a hyperparameter.
$\boldsymbol{v}$	a vector of computed velocity terms per each weight $\boldsymbol{w}$.
$\boldsymbol{w}$	is a vector of all network weights.
t	is a scalar time step, representing the iteration comprising a forward pass for making a prediction, and a backward pass to compute the gradients.

1.2.4 Adagrad

Adagrad (Duchi et al., 2011) aims to improve on gradient descent by effectively adjusting the learning rate per weight based on the history of the gradients for that weight. To accomplish this adagrad accumulates the square of the gradient each time-step, and divides the current gradient by the square root of previous sum of square gradients. This has the beneficial effect of normalizing the learning rate based on the past values. Weights with small gradients will have their effect boosted compared to weights with large gradients. This is particularly important in deep neural networks where early layers will be more significantly impacted by the vanishing gradient problem (Bengio et al., 1994; Pascanu et al., 2013). The downside of Adagrad is that the accumulation of the square gradient in the denominator causes the learning rate to decay over time. This is often cited as a problem, though it can be noted that a simple heuristic solution to that problem would be to increase the global learning rate passed as a parameter to the algorithm.

The Adagrad update is given by the following equations, first describing the non-vectorized form (since Adagrad uses per-weight updates), and then the vectorized form:

$$g_{t,i} = \nabla_w J(w_{t,i}) \tag{1.6}$$

$$w_{t+1,i} = w_{t,i} - \eta \cdot g_{t,i} \tag{1.7}$$

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} \tag{1.8}$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\boldsymbol{G}_t + \epsilon}} \odot \boldsymbol{g}_t \tag{1.9}$$

$g_{t,i}$	the per-weight gradient w.r.t. the loss function.
$w_{t,i}$	each trainable weight in the network indexed by i at time step t.
η	is a scalar learning rate provided to the algorithm as a hyperparameter.

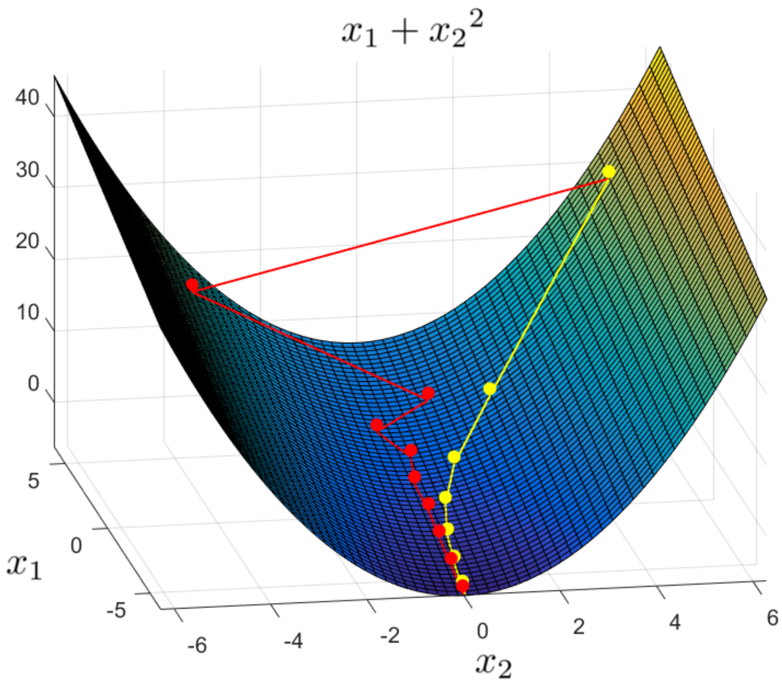


FIGURE 1.3: Adagrad update shown in yellow, compared with the Nestrov update in red.

G_t	$\in \mathbb{R}^{d \times d}$, where d is the number of weights in $\boldsymbol{w}$, is a diagonal matrix where the diagonal elements indexed by i, i are the sum of the square gradients with respect to θ_i up to time step t.
$G_{t,ii}$	is one of the diagonal elements of G.
ϵ	is a small value to avoid numerical issues.
$\boldsymbol{w}$	is a vector of all network weights.
t	is a scalar time step, representing the iteration comprising a forward pass for making a prediction, and a backward pass to compute the gradients.
$\odot$	is the Hadamard product, a.k.a. element-wise vector multiplication.

1.2.5 Adadelta

Adadelta Zeiler (2012) attempts to take the best of Adagrad while eliminating the decaying learning rate. Adadelta does this by allowing the accumulated square gradient term to decay over time, taking only a fraction of the accumulator on each step, which causes the estimate to be biased towards recent updates over earlier updates. Adadelta further maintains a sum of square weight values.

Taking the ratio between decaying square weight values and decaying square gradient values provides a learning rate per weight that adjusts to be smaller for gradients that are large and larger for gradients that are small.

This has a dual benefit of helping train weights that are making slow progress, and keeping them in line, relatively speaking, with weights that have a larger gradient. This also has benefit in that it naturally increases the learning rate in earlier layers where the vanishing gradient problem will be more pronounced (Bengio et al., 1994; Pascanu et al., 2013).

The Adadelta update is given by the following equations:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \Delta\boldsymbol{w}_t \tag{1.10}$$

$$\Delta\boldsymbol{w}_t = -\frac{RMS[\Delta\boldsymbol{w}]_{t-1}}{RMS[\boldsymbol{g}]_t} \odot \boldsymbol{g}_t \tag{1.11}$$

$$RMS[\boldsymbol{w}]_t = \sqrt{E[\Delta\boldsymbol{w}^2]_t + \epsilon} \tag{1.12}$$

$$RMS[\boldsymbol{g}]_t = \sqrt{E[\boldsymbol{g}^2]_t + \epsilon} \tag{1.13}$$

$$E[\boldsymbol{g}^2]_t = \gamma E[\boldsymbol{g}^2]_{t-1} + (1-\gamma)\boldsymbol{g}_t^2 \tag{1.14}$$

γ	is a hyperparameter configured similarly to Momentum with a common default of 0.9.
η	is a scalar learning rate provided to the algorithm as a hyperparameter.
$E[\boldsymbol{g}^2]_t$	is the running average at time step t, E is only a running approximation to $\mathbb{E}$ expectation.
$\boldsymbol{g}$	is the vector of gradients.
$\boldsymbol{g}^2$	is the element-wise square of the gradients $\boldsymbol{g}$.
$\boldsymbol{w}$	is a vector of all network weights.
t	is a scalar time step, representing the iteration comprising a forward pass for making a prediction, and a backward pass to compute the gradients.
$\odot$	is the Hadamard product, a.k.a. element-wise vector multiplication.
ϵ	is a small value to avoid numerical issues.

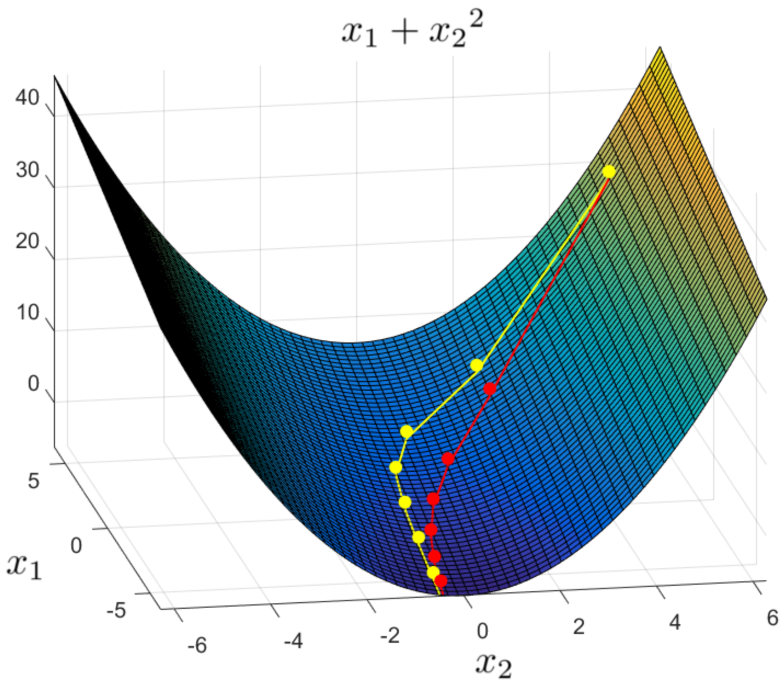


FIGURE 1.4: Adagrad in red compared to Adadelta in yellow. Adadelta is initialized with the gradient sums = 1.0 at start to avoid pathological initialization issues.

1.2.5.1 Adadelta Initialization Issues

Adadelta has a few issues that are identified here. The initialization of the gradient sums is quite critical. The algorithm can be pathologically slow when the gradient sum is initialized to zero. An initialization of 1.0 seems to be reasonable, producing convergence that seems reasonable in the toy example used here. However initializing to a large number such as 10.0 causes the algorithm to exhibit the same zig-zag effect of vanilla stochastic gradient descent, in fact it will converge so slowly as to be irrelevant. Even after 1000 iterations the zig-zag effect remains when the gradient sums are in this poor state. It should be further noted that the zig-zag effect in the x_1 dimension doesn't affect progress in the x_2 direction though, so the per-weight learning rate still ameliorates that problem.

The issues with initialization are particularly important because we experiment with using adadelta's per-weight learning rate in our exponentiated gradient update algorithms (see section 3.7). The initialization issues are visualized in Figure 1.5.

In summary, Adadelta allows us to eliminate the per-weight learning rate, though the algorithm exhibits pathological states under certain conditions. It's hard to know whether these issues are commonly encountered in practice.

1.2.6 Window Grad

Window Grad (Zeiler, 2012) was proposed in the same paper with Adadelta. It's not as widely implemented as Adadelta or other methods. Window Grad leaves out an accumulator of the change in weight values and only utilizes an exponentially decaying accumulation of the gradients. Whereas Adadelta completely eliminates the learning rate, and adjusts the weights to their hypothetical unit value, Window Grad can be thought of as an update in the units of the gradient. Window Grad requires a learning rate to be set. Adadelta can be thought of as the completed extension of the Window Grad idea. Visualization in 1.6.

The Windowgrad update is given by the following equations:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \Delta \boldsymbol{w}_t \tag{1.15}$$

$$\Delta \boldsymbol{w}_t = -\frac{\eta}{RMS[\boldsymbol{g}]_t} \odot \boldsymbol{g}_t \tag{1.16}$$

$$RMS[\boldsymbol{g}]_t = \sqrt{E[\boldsymbol{g}^2]_t + \epsilon} \tag{1.17}$$

$$E[\boldsymbol{g}^2]_t = \gamma E[\boldsymbol{g}^2]_{t-1} + (1-\gamma)\boldsymbol{g}_t^2 \tag{1.18}$$

γ is a hyperparameter configured similarly to Momentum with a common default of 0.9.

η is a scalar learning rate provided to the algorithm as a hyperparameter.

$E[\boldsymbol{g}^2]_t$	is the running average at time step t, E is only a running approximation to $\mathbb{E}$ expectation.
$\boldsymbol{g}$	is the vector of gradients.
$\boldsymbol{g}^2$	is the element-wise square of the gradients $\boldsymbol{g}$.
$\boldsymbol{w}$	is a vector of all network weights.
t	is a scalar time step, representing the iteration comprising a forward pass for making a prediction, and a backward pass to compute the gradients.
$\odot$	is the Hadamard product, a.k.a. element-wise vector multiplication.
ϵ	is a small value to avoid numerical issues.

1.2.7 RMSprop

RMSprop (Hinton) is an unpublished algorithm proposed by Hinton. Ironically the citation commonly used for it is to slide 29 lecture 6 from a lecture by Hinton. RMSprop can be compared very similarly to Adadelta. It attempts to extend Adagrad in a very similar way that Adadelta does. It maintains the per-weight learning rate while eliminating the decaying learning rate inherent in Adagrad.

RMSprop maintains a "cache" of past weight values which decay over time given a decay parameter and accumulates the square gradient. The current gradient is divided by this "leaky" cache to modulate the learning rate per weight. RMSprop maintains a global learning rate parameter that Adadelta gets rid of.

One notable difference that is apparent based on the visualizations provided in this thesis (Figure 1.7) is that RMSprop does not have the same initialization problem that Adadelta has. The visual here shows Adadelta and RMSprop performing almost identically, however it needs to be noted that Adadelta was initialized with a good initial state (1's for the gradient sum).

The RMSprop update is given by the following equations:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{E[\boldsymbol{g}^2]_t + \epsilon}} \cdot \boldsymbol{g}_t \tag{1.19}$$

$$E[\boldsymbol{g}^2]_t = \gamma E[\boldsymbol{g}^2]_{t-1} + (1-\gamma)\boldsymbol{g}_t^2 \tag{1.20}$$

γ	is a hyperparameter configured similarly to Momentum with a common default of 0.9.
η	is a scalar learning rate provided to the algorithm as a hyperparameter.
$E[\boldsymbol{g}^2]_t$	is the running average at time step t, E is only a running approximation to $\mathbb{E}$ expectation.
$\boldsymbol{g}$	is the vector of gradients.
$\boldsymbol{g}^2$	is the element-wise square of the gradients $\boldsymbol{g}$.
$\boldsymbol{w}$	is a vector of all network weights.
t	is a scalar time step, representing the iteration comprising a forward pass for making a prediction, and a backward pass to compute the gradients.
ϵ	is a small value to avoid numerical issues.

1.2.8 Adam

Adam (Kingma and Ba, 2014) is another adaptive optimization method which applies a momentum term in a new way. Adam is often cited as the best all around stochastic optimization method to start with. Adam applies the concept of momentum as well as adaptive learning rates per weight. Adam operates in much the same way as RMSprop in that it maintains a history of square gradients that it uses to adapt the per-weight learning rate. Adam further accounts for the fact that the initialization causes a bias towards zero and accounts for this by computing the first and second moments correcting for the bias. Adam updates are visualized in Figure 1.8.

The Adam update is given by the following equations:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon} \cdot \hat{\boldsymbol{m}}_t \tag{1.21}$$

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1 - \beta_1} \tag{1.22}$$

$$\hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1 - \beta_2} \tag{1.23}$$

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1) \boldsymbol{g}_t \tag{1.24}$$

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2) \boldsymbol{g}_t^2 \tag{1.25}$$

$\boldsymbol{m}$	an exponentially decaying average of past gradients
$\hat{\boldsymbol{m}}$	the bias-corrected first moment
$\boldsymbol{v}$	an exponentially decaying average of past gradients squared
$\hat{\boldsymbol{v}}$	the bias-corrected second moment
$\boldsymbol{w}$	is a vector of all network weights.
β_1	a hyperparameter with a recommended default of 0.9
β_2	a hyperparameter with a recommended default of 0.999
t	is a scalar time step, representing the iteration comprising a forward pass for making a prediction, and a backward pass to compute the gradients.
η	is a scalar learning rate provided to the algorithm as a hyperparameter.
ϵ	is a small value to avoid numerical issues.

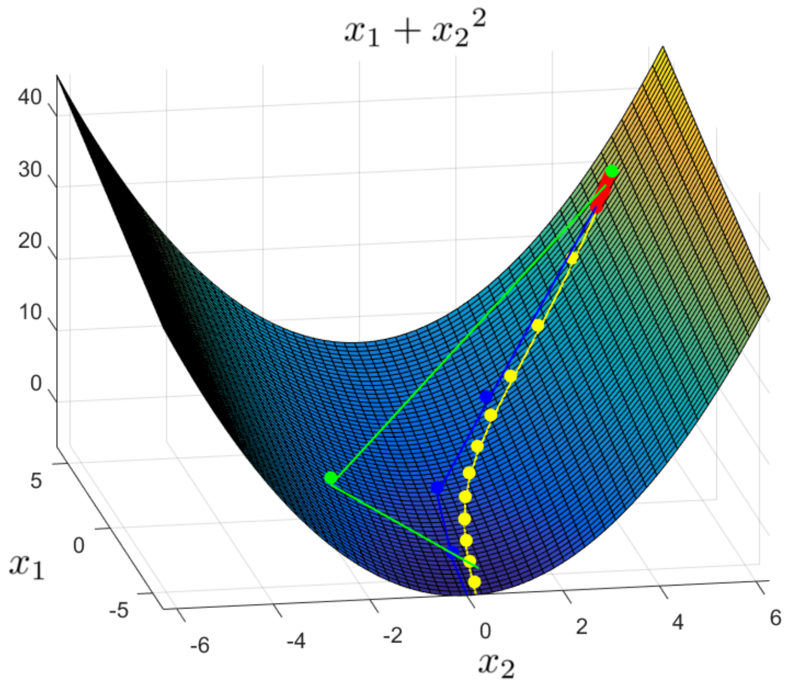


FIGURE 1.5: Adadelta progress using 4 different initializations of the gradient sum. In red the gradient sum is initialized to 0.0, which is the default implementation of Adadelta by the papers author in ConvnetJS. This initialization has a pathological issue in this 2D example causing convergence to be extremely slow due to a slowly accumulating x_2 sum, and quickly accumulating gradient sum. In yellow and blue Adadelta is initialized with a gradient sum of 0.1 and 1.0, both reasonable initializations which converge to within 0.1 of the x_2 axis in 25 steps and 6 steps respectively. The green line initializes the gradient sum with 5.0 and converges within 41 steps while exhibiting SGD's oscillation back and forth in the valley. Higher initialization values exacerbate the oscillation effect for longer time periods.

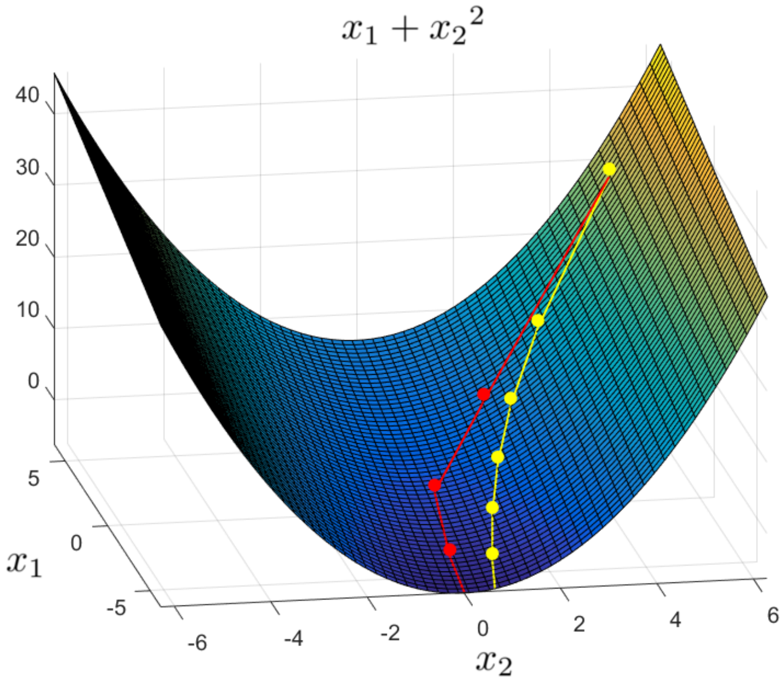


FIGURE 1.6: Adadelta in red vs. Window Grad in yellow, initialized with a "good" sum-square gradient initialization of 1.0 to avoid a pathologically slow start.

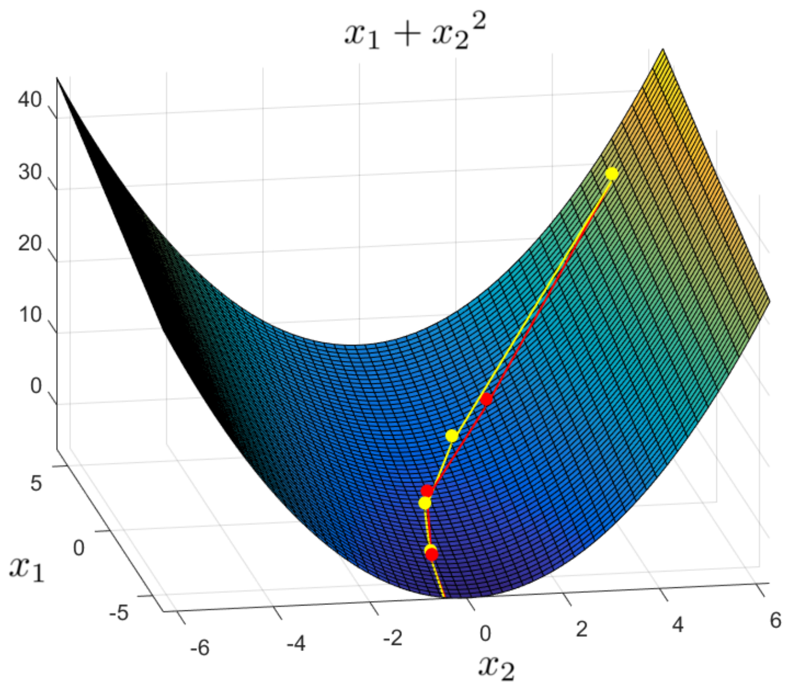


FIGURE 1.7: In red, Adadelta with a gradient sum initialization of 1's (e.g. a "good" initialization), and RMSprop in yellow. Both perform very similarly on this 2 dimensional example, and long term convergence is quite similar: Both converge on the x dimension in 6 steps, though after 100 iterations RMSprop had moved down to -100 in the y direction, vs. -300 for Adagrad, so Adagrad took larger steps down the chasm.

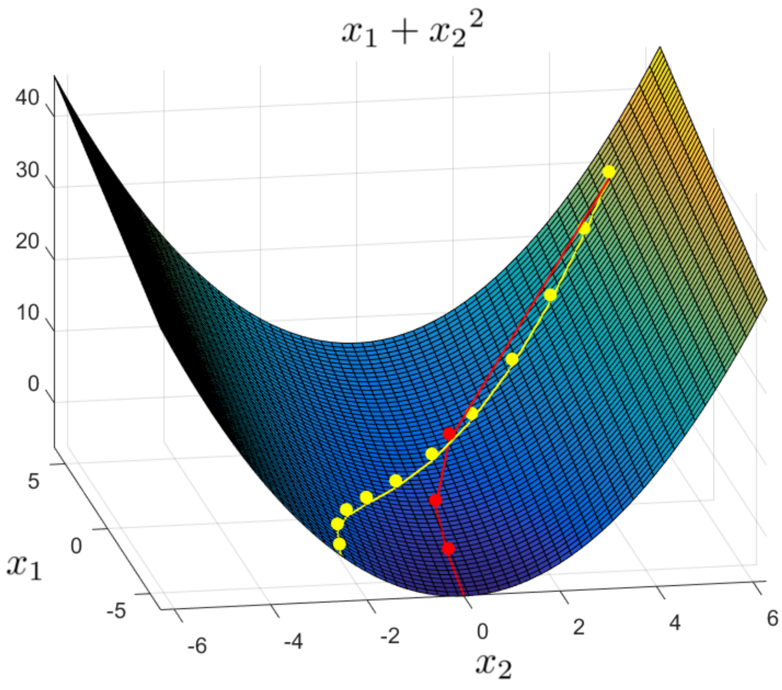


FIGURE 1.8: RMSprop shown in red with Adam updates shown in yellow. Although the RMS prop visualization appears to behave better in this case, the true test of an optimizer in high dimensional space is how well it performs on test set evaluation. Low dimensional visuals can help in understanding the behavior, but are not necessarily a clear indicator of optimal performance.

Chapter 2

Exponentiated Gradient $\pm$ Update Algorithm

In this Chapter we introduce the Exponentiated Gradient $\pm$ (henceforth EG$^{\pm}$) (Kivinen and Warmuth, 1995, 1997) algorithm, and its predecessor the Exponentiated Gradient algorithm (henceforth EG). EG was introduced by Manfred Warmuth and Jyrki Kivinen in 1995 in the context of linear online predictors. The algorithm has been introduced for use in neural networks formally by Srinivasan et al. (2002), for which we extend the analysis to a more modern context, and informally discussed in Langford (2007).

EG$^{\pm}$ generalizes the Exponentiated Gradient (EG) algorithm which maintains a probability distribution over weights. EG does not allow for weights to change from positive to negative or vice versa. EG$^{\pm}$ operates by doubling the number of weights and performs an EG update to both positive and negative components of the weight.

We begin by introducing the EG algorithm first.

2.0.1 Exponentiated Gradient

The Exponentiated Gradient (EG) algorithm performs an update by computing a multiplicative factor for each weight and multiplying that factor into each individual weight, then re-normalizing by dividing by the total sum of the weights. A global learning rate parameter that is used to scale the update factor, and functions the same as it does with SGD and its variants.

The Exponentiated Gradient algorithm is derived by trading off the relative entropy with the loss ameliorated by the learning rate that is defined by the following minimization problem: $\boldsymbol{w}_{t+1} = \underset{\boldsymbol{w}\, s.t. \sum w_i = 1}{\operatorname{argmin}} \sum \left(w_i \ln \frac{w_i}{w_{t,i}} \right) + \eta J(\boldsymbol{w})$ and this results in the update algorithm given below. EG is motivated by using a relative entropy regularization.

$$w_{t+1,i} = \frac{w_{t,i} \cdot e^{(-\eta \nabla_w J(w_{t+1,i}))}}{\sum_{j=1}^{N} w_{t,j} \cdot e^{(-\eta \nabla_w J(w_{t+1,j}))}} \approx \frac{w_{t,i} \cdot e^{(-\eta \nabla_w J(w_{t,i}))}}{\sum_{j=1}^{N} w_{t,j} \cdot e^{(-\eta \nabla_w J(w_{t+1,j}))}} \tag{2.1}$$

$w_{t,i}$	is a single weight at time step t indexed by i.
$w_{t,j}$	is a single weight at time step t indexed by j.
N	number of weights in $\boldsymbol{w}$
i	$i \in 1, ..., N$, per each weight in $\boldsymbol{w}$
j	$j \in 1, ..., N$, per each weight in $\boldsymbol{w}$
t	is a scalar time step, representing the iteration comprising a forward pass for making a prediction, and a backward pass to compute the gradients.
$\nabla_{\boldsymbol{w}} J$	the gradient of the weights with respect to the loss function. This is the gradient calculated by backprop.
η	is a scalar learning rate provided to the algorithm as a hyperparameter.

2.0.2 Exponentiated Gradient $\pm$

EG$^{\pm}$ is a generalization of the Exponentiated Gradient (EG) algorithm which allows weights to take on positive or negative values. EG$^{\pm}$ achieves this by maintaining two weight vectors, $\boldsymbol{w}^{+} = \{x \in \mathbb{R}^d | x \geq 0\}$ and $\boldsymbol{w}^{-} = \{x \in \mathbb{R}^d | x \geq 0\}$, where d is the number of weights in the neural network, represented by $\boldsymbol{w} = \{x \in \mathbb{R}^d\}$. The basic EG update is applied to the each weight vector separately, then scaled by a U parameter, and normalized. The neural network weights take on the value $\boldsymbol{w} = \boldsymbol{w}^{+} - \boldsymbol{w}^{-}$.

EG$^{\pm}$ shares some properties with SGD, Figure 2.1 demonstrates a simple plot where EG$^{\pm}$ and SGD perform the same updates given a fixed gradient as the weight value updates from 0.1 to -0.1. In later sections we will visualize this (Figure 2.4) and show comparable performance on adversarial samples in Section 3.14.

The EG$^{\pm}$ update is given by the equations:

$$w^{+}_{t+1,i} = U \cdot \frac{w^{+}_{t,i} r^{+}_{t,i}}{\sum_{j=1}^{N} w^{+}_{t,j} r^{+}_{t,j} + w^{-}_{t,j} r^{-}_{t,j}} \tag{2.2}$$

$$w^{-}_{t+1,i} = U \cdot \frac{w^{-}_{t,i} r^{-}_{t,i}}{\sum_{j=1}^{N} w^{+}_{t,j} r^{+}_{t,j} + w^{-}_{t,j} r^{-}_{t,j}} \tag{2.3}$$

where

$$r^{+}_{t,i} = \exp\left(-\eta \nabla_{w_{t,i}} J(w_{t,i})\right) \tag{2.4}$$

$$r^{-}_{t,i} = \exp\left(\eta \nabla_{w_{t,i}} J(w_{t,i})\right) = \frac{1}{r^{+}_{t,i}} \tag{2.5}$$

U	U is a scaling parameter that is provided as a hyperparameter, typical values are 20 to 80.
$w_{t,i}$	A single EG$^{\pm}$ weight at update step t, indexed by i.
$w_{t,i}$	is a weight used by the neural network at time t indexed by i.
$\nabla_{w_{t,i}} J$	The gradient of the loss function with respect to the combined weight value $w_{t,i}$. This is the gradient computed in backprop, and is with respect to the combined weight value, not $w^{+}_{t,i}$ or $w^{-}_{t,i}$ individually.
t	is a scalar time step, representing the iteration comprising a forward pass for making a prediction, and a backward pass to compute the gradients.
η	is a scalar learning rate provided to the algorithm as a hyperparameter.
i	index of the weight. The update equations describe the update per weight.
j	index over the set of weights being normalized against, the set of weights being normalized against depends on the normalization method employed (per weight/neuron/layer/network). See Section 2.0.2.3
N	number of weights being normalized against, the set of weights being normalized against depends on the normalization method employed (per weight/neuron/layer/network). See Section 2.0.2.3

2.0.2.1 Memory footprint and computation cost

EG$^{\pm}$ requires two values per neural network weight, though the update is still in $O(n)$ time, and is comparable in computation cost to other optimizers such as Adam. In practice it's most reasonable that the update be implemented by

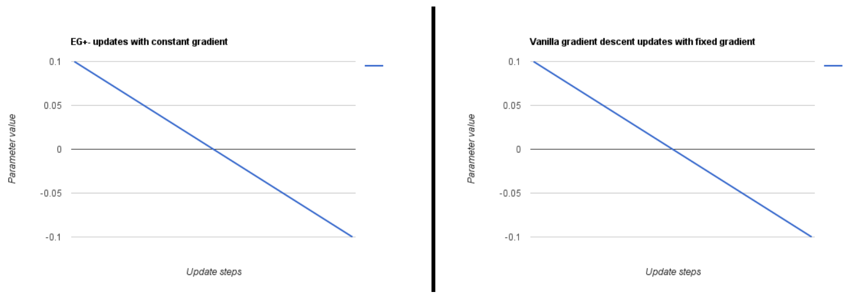


FIGURE 2.1: We take a toy 1D example and plot 155 update steps of EG$^{\pm}$ and of SGD. In both cases we start at 0.1 and hold the gradient fixed at 0.05. SGD has a learning rate of 0.08, and EG$^{\pm}$ uses 0.004 with a U scaling parameter of 20 (see 2.0.2 for details). Given these parameters both EG$^{\pm}$ and SGD perform nearly identical updates.

maintaining $2n$ additional weights for the two vectors, $\boldsymbol{w}^+$ and $\boldsymbol{w}^-$, in addition to n weights of the network which represent the sum of $\boldsymbol{w}^+$ and $\boldsymbol{w}^-$. EG$^{\pm}$ requires that the two vectors $\boldsymbol{w}^+$ and $\boldsymbol{w}^-$ be independent of the weights used in feedforward and backprop iterations. In forward/back propagation, the weights will be the difference of $\boldsymbol{w}^+ - \boldsymbol{w}^-$, so the actual memory footprint of a typical implementation is $3n$ weights. While this is a cost, it's typically a small fraction of the total memory footprint of practical neural networks. Additionally, technically feasible to reduce the memory footprint in an optimized implementation because the weight values are restricted to be positive and limited by the U scaling parameter as discussed in the next section. Hence storing both weights in one vector is possible at the expense of additional CPU cycles (this is outside the scope of this paper, suggested further reading `https://en.wikipedia.org/wiki/Pairing_function`).

2.0.2.2 U Scaling

EG$^{\pm}$ requires three parameters to be provided, learning rate, which is used as is typical for update algorithms; the normalization method to use; and a U scaling parameter. The U scaling parameter is critical to the algorithm and has subtle effects that need to be understood.

The $\boldsymbol{w}^+$ and $\boldsymbol{w}^-$ vectors maintained by EG$^\pm$ are updated and normalized (Section 2.0.2.3) to sum to 1 each time step t. After the normalization they are scaled by a parameter U. Typical values of U are in the range of 20 to 80 as discussed and visualized in Section 3.2. The U scaling parameter has two important effects:

- Numerical stability: we need the difference between $\boldsymbol{w}^+$ and $\boldsymbol{w}^-$ to be well within rounding error, in particular because neural networks typically use 32 bit floating point precision, and it's been shown that 16, and even 8 bit floating point precision can be used (Srinivasan et al., 2002; Deng et al., 2015).

- More important than numerical stability is the fact that the U scaling parameter limits the maximum or minimum range that a weight can take. If U=20, the weight values can range from $[-20, 20]$. This is visualized in Figure 2.2.

This second bullet bears further discussion. This limitation might be reasonably compared to Max-norm regularization which has been shown to have benefit in neural networks (Srebro and Shraibman, 2005; Srivastava et al., 2014). U scaling will cause the progression of a weight towards its max to asymptote at the U scaling parameter value. A beneficial difference between U scaling and Max-norm regularization would be that U-scaling will approach its maximum smoothly. As occurs in the softmax function, the value assigned to a weight approaching the U scaling parameter will be redistributed to other weights in the network via the normalization process.

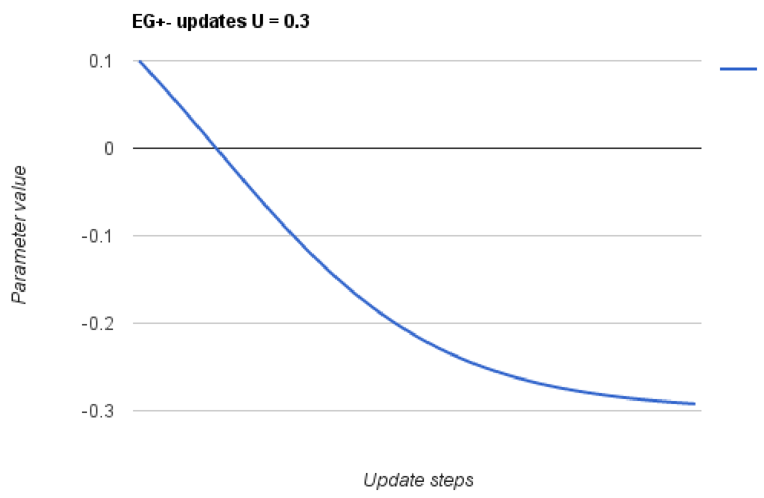


FIGURE 2.2: 1D example of the U scaling limitation, the x-axis is from 1 to 155 update steps, and the y-axis shows the weight value. This visual performs EG+- updates with a gradient of 0.5 for each step, and a beginning weight value of 0.1, learning rate of 0.1, and U parameter of 0.3. The weight is bound between +- 0.3 with updates smoothly approaching the asymptote.

It should be noted, however, that in practical examples explored in this paper, the U scaling parameter was typically larger than weight values observed in the trained networks. It should also be noted that depending on the implementation, the U scaling limitation may also apply equally to bias units as they do to the weights. These issues are explored in more detail in Section 3.2 with visuals and analysis of related use cases.

2.0.2.3 Normalization method

The normalization constant for EG$^{\pm}$ in Equation 2.2 and 2.3 is defined as $\sum_{j=1}^{N} w_{t,j}^{+} r_{t,j}^{+} + w_{t,j}^{-} r_{t,j}^{-}$. We experiment with 4 forms of normalization constant in this work: (1) Per-weight normalization; (2) Per-neuron normalization; (3) Per-layer normalization; and (4) Per-network normalization. These methods differ over which weights are included in the normalization process. A visual aid

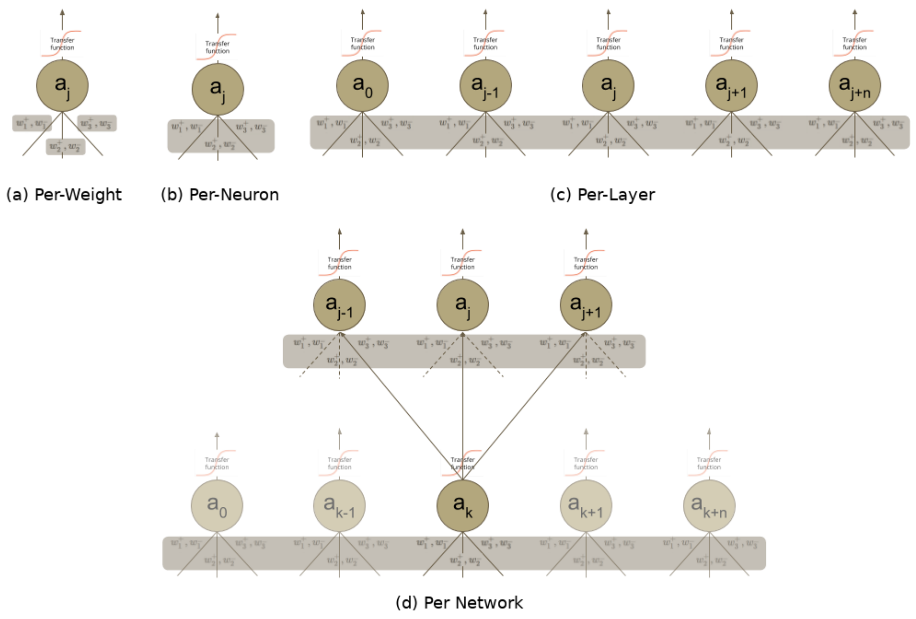


FIGURE 2.3: Visual aid showing the weights that EG$^{\pm}$ is normalized against in the case of (a) per-weight normalization (each weight normalized only against its own w^+ and w^-), (b) per-neuron normalization (each weight normalized against all weights input to a neuron), (c) per-layer (each weight normalized against all weights in a layer), and (d) per-network (each weight normalized against all weights in the network).

depicting the set of weights each method normalizes over is shown in Figure 2.3.

Per-weight normalization is normalized by only the $\boldsymbol{w}^+$ and $\boldsymbol{w}^-$ values per each weight, there is no normalization across different weights in the network. In this case $N = 1$, the summation operator is an extraneous symbol.

Per-neuron normalization is normalized such that each weight is normalized against all weights feeding into that neuron. In the case of a fully connected neural network layer each neuron is the weighted sum of the neurons in the layer before it. The summation is over the weights of the neurons that input into each neuron.

In the case of convolutional network per-neuron normalization is per output filter of the convolution operation. The weight matrix of a convolutional neural network layer, by common convention, has the shape [`kernel_height`, `kernel_width`, `in_channels`, `out_channels`]. Per neuron normalization sums all weights per each `out_channel` such that you end up with a vector of shape [`out_channels`].

A general vectorized solution for per-neuron normalization is implemented in the Tensorflow reference implementation of EG$^{\pm}$ and relies on convention that is used in Tensorflow. Other frameworks may implement different conventions. The general vectorized solution sums each weight matrix across all but the last dimension, producing a final vector of the shape of the last dimension of the weight matrix. The vectorized solution is discussed below in the context of fully connected layers, convolutional layers, and recurrent neural networks.

- *Fully connected layers* have a weight matrix of shape [`neurons_in`, `neurons_out`]. This is the most trivial case, a summation of such a matrix which preserves the `neurons_out` dimension produces a value for each neuron which is the sum of all inputs to that neuron.

- *Convolutional layers* have a weight matrix of shape [`kernel_height`, `kernel_width`, `in_channels`, `out_channels`], which is a common convention which may not hold under all frameworks. The vectorized solution sums over the first 3 dimensions which cover the kernel height and width, and input channels, which represent all input weights per output channel.

- *Recurrent neural networks* such as LSTMs and GRUs simply consist of multiple fully connected networks linked together in non trivial ways. An LSTM for example has four fully connected network operations. Each of these operations is typically implemented using four 2D weight matrices, each of which is just a fully connected network. Under this convention the same process that applies to a fully connected network applies to the RNN.

Per-layer normalization normalizes across all weights in a particular layer. As an implementation detail this method is implemented per-variable because each layer in a neural network is typically defined using a single weight tensor. This is a common convention implemented identically in all frameworks as far as we are aware, but this normalization method also depends on the convention. It should be noted that some variables exist that are not explicitly network weights or biases, such as is the case with trainable batch normalization variables. In the case of batch normalization the trainable weights are implemented in a vector (a 1-D tensor) which would default to being optimized as per-weight normalization because there are no dimensions to sum over.

In the case of per-layer normalization the normalizing constant is summed across all weights of a particular network layer. In most cases this is appropriately named per-layer normalization, but in the case of batch normalization it's more apt to refer to it by the more specific term per-variable normalization where the term layer doesn't logically apply. This document will use the per-layer terminology henceforth for simplicity.

Per-network normalization normalizes across all trainable weights in the network. Note that in the reference implementation this would include variables such as bias units, batch normalization, or any other trainable variables. This is effectively just per-layer normalization summed across all layers.

2.0.3 EG$^{\pm}$ update algorithm

This section concerns actual implementations of EG$^{\pm}$, including a detailed description of the update algorithm, and an example reference implementation.

The update algorithm for EG$^{\pm}$ is detailed in Algorithm 1.

Algorithm 1: EG$^{\pm}$ update procedure using per-neuron normalization.

```
Data: U hyperparameter
      $\eta$ learning rate hyperparameter
      Neural network trainable weights $\boldsymbol{w}_{t-1}$
      EG$^{\pm}$ weights $\boldsymbol{w}^{+}_{t-1}$, and $\boldsymbol{w}^{-}_{t-1}$
      A gradient per neural network weight $\nabla_{\boldsymbol{w}}$
      $\boldsymbol{w}, \boldsymbol{w}^{+}, \boldsymbol{w}^{-}$ are all initialized as per Algorithm 2 at $t = 0$.
Result: Updated EG$^{\pm}$ positive and negative weight vectors $\boldsymbol{w}^{+}_{t}$ and $\boldsymbol{w}^{-}_{t}$,
        and neural network weights $\boldsymbol{w}_t$
for $t = 1$ to convergence do
    for each weight: $w_{t-1,i}$, $w^{+}_{t-1,i}$, $w^{-}_{t-1,i}$ do
        $r^{+}_{i} \leftarrow \exp(-\eta \nabla_{w_{t-1,i}})$
        $r^{-}_{i} \leftarrow \frac{1}{r^{+}_{i}}$
        $w^{+}_{t,i} \leftarrow U \frac{w^{+}_{t-1,i} r^{+}_{i}}{w^{+}_{t-1,i} r^{+}_{i} + w^{-}_{t-1,i} r^{-}_{i}}$
        $w^{-}_{t,i} \leftarrow U \frac{w^{-}_{t-1,i} r^{-}_{i}}{w^{+}_{t-1,i} r^{+}_{i} + w^{-}_{t-1,i} r^{-}_{i}}$
        $w_{t,i} \leftarrow w^{+}_{t,i} - w^{-}_{t,i}$
    end
end
```

2.0.4 Concrete Implementations

Three implementations of EG$^{\pm}$ were build in the process of creating this thesis.

The first in Matlab (Parks, 2016), which is a feed forward neural network implementation written from the ground up based on the online book Neural Networks and Deep Learning (Nielsen, 2015). This code was used to compare EG$^{\pm}$ in depth against SGD using a variety of regularization methods (see Section 3).

The second implementation was added to ConvnetJS (Karpathy), a feed forward and convolutional framework built on JavaScript which runs in the browser. This framework was chosen because it provides excellent visualizations and comparison with other update algorithms which were used as a comparison and visualization platform for the algorithm.

A version of ConvnetJS will be hosted at UCSC until and unless the EG$^{\pm}$ enhancement is accepted into the mainline code, and will be accessible via https://goo.gl/mzosa6, and https://goo.gl/68hIZb.

The third implementation is considered the reference implementation for EG$^{\pm}$ and is written in Python using the Tensorflow framework. It is hosted on github at https://github.com/davidparks21/eg_plusminus_optimizer. This implementation was used to run tests of EG$^{\pm}$ on large residual networks on the GPU.

There are a few notable differences between the implementations. Most importantly is how they handle updating the bias units. In the Matlab implementation EG$^{\pm}$ is applied to the weights, but it is not applied to the bias units. Bias units are updated with standard SGD. The reason for this is the U parameter scaling. The bias units are not usually regularized (see chapter 7-regularization in the book Deep Learning, Goodfellow et al.), the bias is responsible for offsetting the function from the origin, hence it should be acceptable for it to grow arbitrarily large without negatively impacting the algorithms ability to learn. The bias does not represent an over reliance on any one feature. However in this thesis we will identify reasons why this isn't a concern in practice (see Section 3.3).

While it was possible to separate weight and bias updates in the Matlab implementation, the implementation of ConvnetJS follows a more rigid structure that is common to most frameworks, and is the same in the Tensorflow reference implementation. In those frameworks the weights are presented as a single vector of values, it's non-trivial to identify weights vs. bias units. In these frameworks we apply EG$^{\pm}$ to both weights and biases. In doing so we also analyze the U scaling issues with regard to the bias units (3.2) and find the issue unlikely to be concerning in practice.

Another difference to note is in weight initialization between the implementations. Weight initialization in the Matlab code normalizes the sum of inputs to the neurons, see (Nielsen, 2015; Glorot and Bengio, 2010). ConvnetJS initializes weights using a zero mean, unit variance distribution. In Tensorflow the weight initialization depends on the model, we use the Xavier Initializer (Glorot and Bengio, 2010) in testing for this thesis. On the grand scheme of things this difference is minor, though the better weight initialization has been shown to produce slightly better final results, and is only mentioned for completeness.

2.0.4.1 EG$^{\pm}$ code from ConvnetJS

A demonstration of EG$^{\pm}$ as implemented in the online deep learning framework ConvnetJS (Karpathy) is shown in Listing 2.1. This implementation updates each weight, one by one, in a loop (which is somewhat inefficient, but useful for demonstration purposes).

2.0.4.2 EG$^{\pm}$ Initialization

The positive and negative weights for EG$^{\pm}$ must be initialized properly. Given a random weight initialization value, the correct initialization of the EG$^{\pm}$ positive and negative weight values is given by the algorithm below. This initialization method was applied to all normalization methods covered in 2.0.2.3. An alternate iterative initialization method was experimented with, but did not work, and is omitted from further discussion. The neural network weight initialization of $\boldsymbol{\theta}$ can follow any standard practice applied to neural networks, such as small random normal initialization or, for example, Xavier initialization as proposed by Glorot and Bengio (2010).

LISTING 2.1: ConvnetJS implementation of EG$^\pm$ with per-neuron normalization, Equations 2.2 to 2.5

```
# Compute updates, gij is the gradient for neuron [i,j]
var rpos = Math.exp(-this.learning_rate * gij);
var rneg = 1 / rpos;
var unscaled_pos = this.w_pos[i][j] * rpos;
var unscaled_neg = this.w_neg[i][j] * rneg;
var normalize = unscaled_pos + unscaled_neg;
# Save weights
this.w_pos[i][j] = this.U * (unscaled_pos / normalize);
this.w_neg[i][j] = this.U * (unscaled_neg / normalize);
W[j] = this.w_pos[i][j] - this.w_neg[i][j];
```

`gij`	The gradient with respect to the weight being updated.
`learning_rate`	The learning rate hyperparameter.
`w_pos`	An array of positive weights maintained by the EG$^\pm$ algorithm.
`w_neg`	An array of negative weights maintained by the EG$^\pm$ algorithm.
`U`	The U scaling hyperparameter.
`W[j]`	The updated weight value.

2.0.5 Visualization

To finish off our introduction to EG$^\pm$ we provide the same visualization of how it would handle the same 2D case we presented for all of the other update algorithms, see Figure 2.4.

It is interesting to note that EG$^\pm$ behaves in essentially the same way as vanilla SGD in this toy case. In this respect it doesn't take into account how the

Algorithm 2: EG$^\pm$ initialization procedure for all normalization methods

```
Data: U hyperparameter, randomly initialized neural network weights $\boldsymbol{w}$,
      two uninitialized EG$^\pm$ weight vectors $\boldsymbol{w}^+$ and $\boldsymbol{w}^-$ the same shape
      as $\boldsymbol{w}$
Result: EG$^\pm$ positive and negative weight vectors $\boldsymbol{w}^+$ and $\boldsymbol{w}^-$
for each neural network weight: $w_i$ do
    $w_i^+ \leftarrow (U + w_i)/2$
    $w_i^- \leftarrow (U - w_i)/2$
end
```

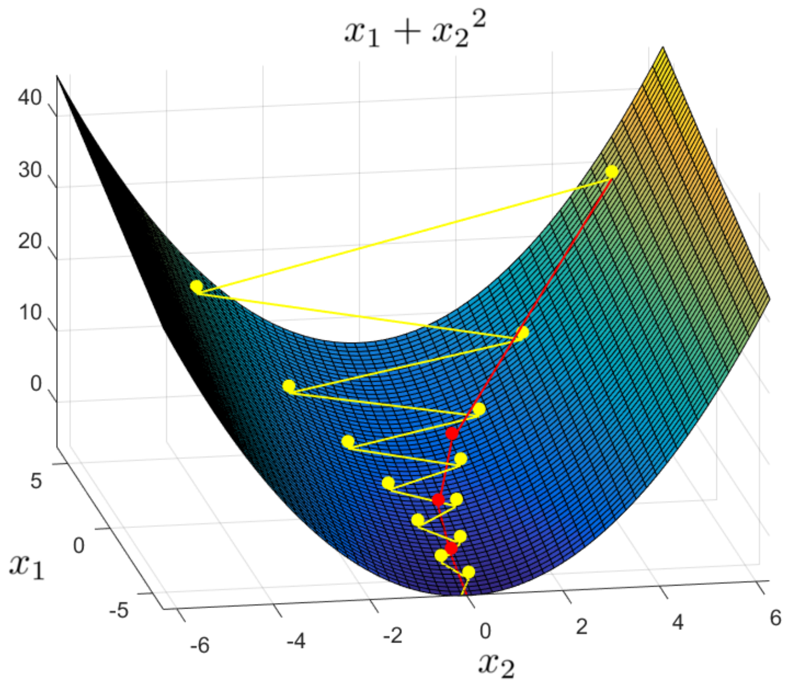


FIGURE 2.4: RMSprop shown in red with EG updates with per-weight normalization shown in yellow.

gradient changes over time. This isn't to say that EG performs exactly the same update as SGD, or that it will perform the same in practice, we will show many ways in which it doesn't, both for better and worse, and we will also experiment with per-weight learning rates for EG (see Section 3.7).

Chapter 3

Findings and conclusions

This section concerns results and experiments performed with $\text{EG}^{\pm}$ compared to SGD and other algorithms. In these experiments we utilize two datasets, the MNIST handwritten digits dataset (Lecun et al., 1998) and CIFAR10 (Krizhevsky, 2009) image classification dataset.

3.1 Summary of findings

Some of the early experimentation with $\text{EG}^{\pm}$ focused on comparing $\text{EG}^{\pm}$ to SGD and comparing how $\text{EG}^{\pm}$ performs on its own with regards to SGD using L1 and L2 regularization. In general cases L1 and L2 regularization with SGD out performed $\text{EG}^{\pm}$. With one particularly notable exception: when training with many features of random noise, $\text{EG}^{\pm}$ clearly out performs SGD (section 3.5).

Later experimentation on $\text{EG}^{\pm}$ focused on comparing it to many other optimization algorithms in common use. In this setting $\text{EG}^{\pm}$ was applied on even footing with other algorithms and L2 regularization was applied to the loss of $\text{EG}^{\pm}$ as well as other algorithms. In this setting $\text{EG}^{\pm}$ performed at the top of the class on a 2 layer MNIST dataset in Karpathy's ConvnetJS deep learning environment (Section 3.6).

We experimented with extending EG$^\pm$ by adding per weight learning rates. In this work we see obvious improvements in how EG$^\pm$ optimizes a manufactured valley problem, but we don't see that extend to a practical example, with the adaptive version performing very similarly to the non adaptive version (Section 3.7).

In the Tensorflow reference implementation of EG$^\pm$ we experimented with EG$^\pm$ on a near state of the art large 32 block residual neural network trained on the cifar10 dataset. In this context we find that EG$^\pm$ performs nearly, but not quite as well as other optimizers. We do find that EG$^\pm$ performs better than most other optimizers on adversarial examples, but this result is also replicated using vanilla SGD and the common link appears to be that momentum based optimizers perform worse on the adversarial examples we tested than do non momentum based optimizers (EG$^\pm$, and vanilla SGD).

3.2 U parameter scaling

We introduced U scaling in 2.0.2.2 in discussing the EG$^\pm$ algorithm. That section includes discussion of how the U parameter applies to EG$^\pm$, in particular that it achieves a form of gradient clipping by limiting the size of the weights absolute value to be no larger than the hyperparameter U.

In the next Section (3.3) we provide a visualization of the distribution of weights and biases in a fully trained convolutional neural network using the CIFAR10 dataset (Krizhevsky, 2009), trained by EG$^\pm$ and another trained with the Nesterov gradient update algorithm. Skipping ahead to those visuals you will note that the scale of weights and biases remains reasonably close to zero. The weights are close to a normal distribution while the biases appear more uniformly distributed. In neither case do the weights or biases range into values that would be affected by a U scaling parameter of 20 or more.

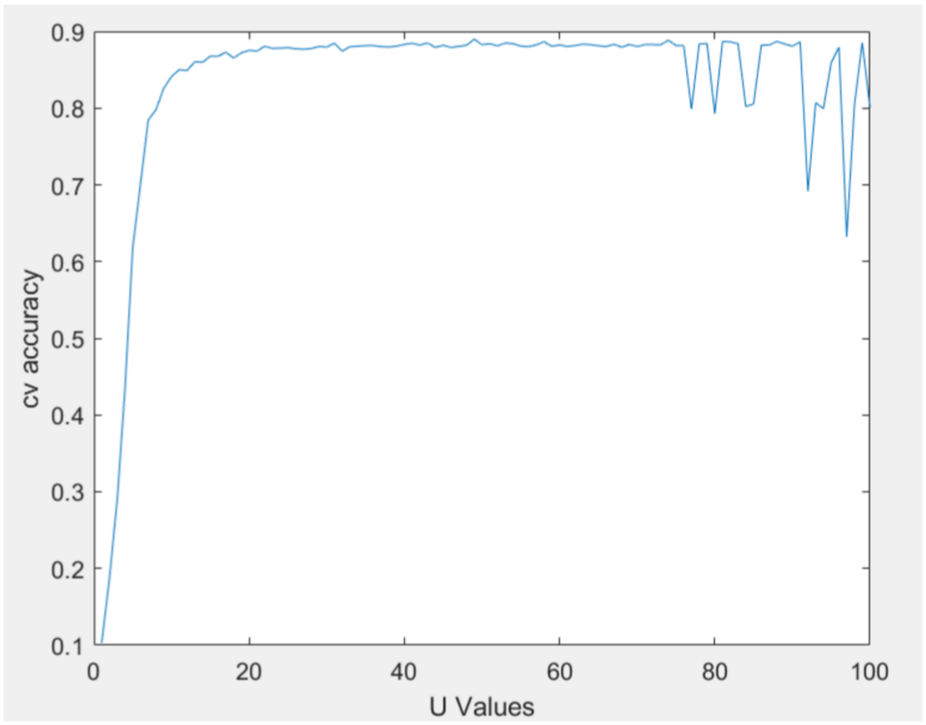


FIGURE 3.1: Accuracy for MNIST dataset when training with varying U parameters from 1 to 100 using per-neuron normalization.

We further analyze another dataset which might reasonably be expected to have large biases in which our output is a regression with outputs in the range $[-60, +60]$. However even in this case the biases do not extend beyond ± 4. So in real world cases we don't see U scaling causing an issue clipping the bias units when it's set to a value that is typically found via a hyperparameter search.

The selection of a good U parameter was one of the first points of interest. Using the MNIST dataset (Lecun et al., 1998), we compare U parameters from 1 to 100 and find that U parameter values starting at 30 to 70 produce consistent results. Note that the results listed here utilized a simplified 1k MNIST training set on fully connected feed forward network with 1 hidden layer.

It is important to note that the selection of the U parameter will have an effect on the results, and thus it's necessary to do a proper hyperparameter search to

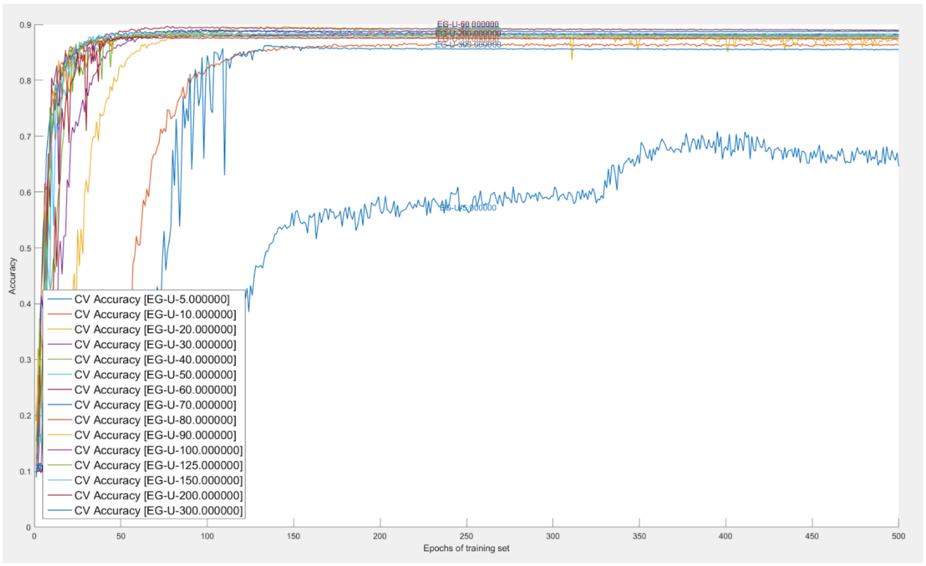


FIGURE 3.2: Test set results based on training a 5k subset of the MNIST dataset using different U scaling parameter values ranging from 5 to 300 and per-neuron normalization.

identify a good choice. Figure 3.2 demonstrates this by training $EG^{\pm}$ on the MNIST dataset using a wide range of U parameters.

3.3 Distribution of weights produced by EG+- vs. Nesterov in a CNN

This section presents the distribution of weights and biases result from training a 3 layer convolutional neural network using both $EG^{\pm}$ and Nesterov optimization. By plotting a histogram of the weights (fig 3.3) and another for the biases (fig 3.4), we can see that the weights and biases follow a very similar distribution, regardless of whether they're trained using $EG^{\pm}$ vs. Nesterov.

This network was trained using ConvnetJS and the Cifar10 dataset and is discussed in more detail in Section 3.12.

It is interesting to note that the distribution of weights takes on a clearly Gaussian form with the mean slightly below zero. And it's particularly useful

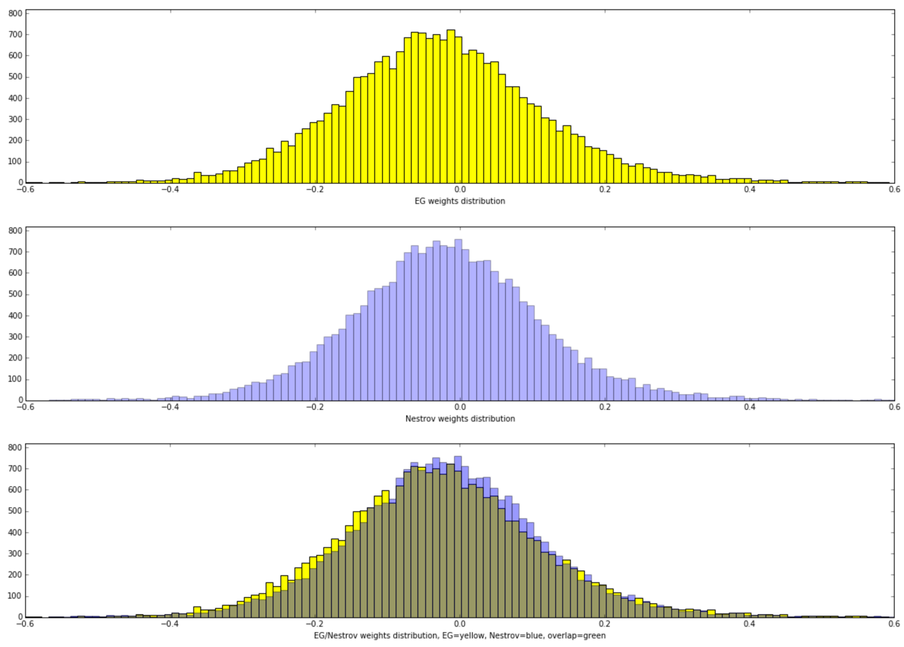


FIGURE 3.3: Histogram of weight values of a convolutional neural network trained on CIFAR10 and optimized by EG$^{\pm}$ (top) and Nesterov (middle), and the two overlapped (bottom).

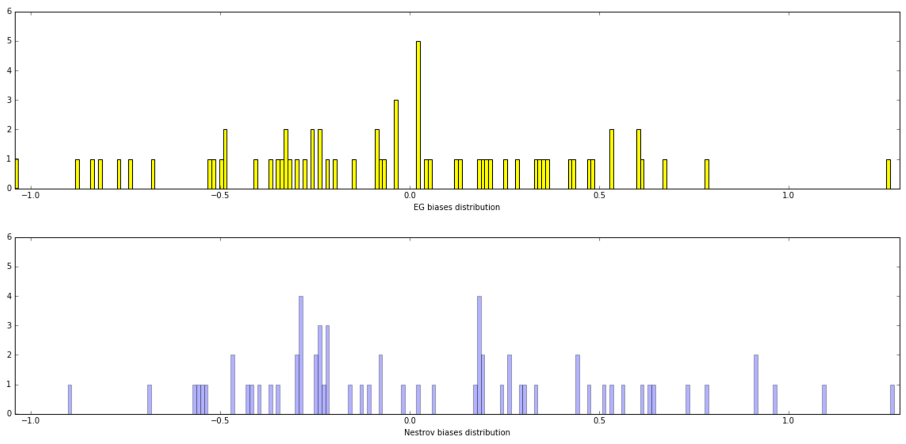


FIGURE 3.4: Histogram of bias values of a convolutional neural network trained on CIFAR10 and optimized by EG$^{\pm}$ (top) and Nesterov (bottom).

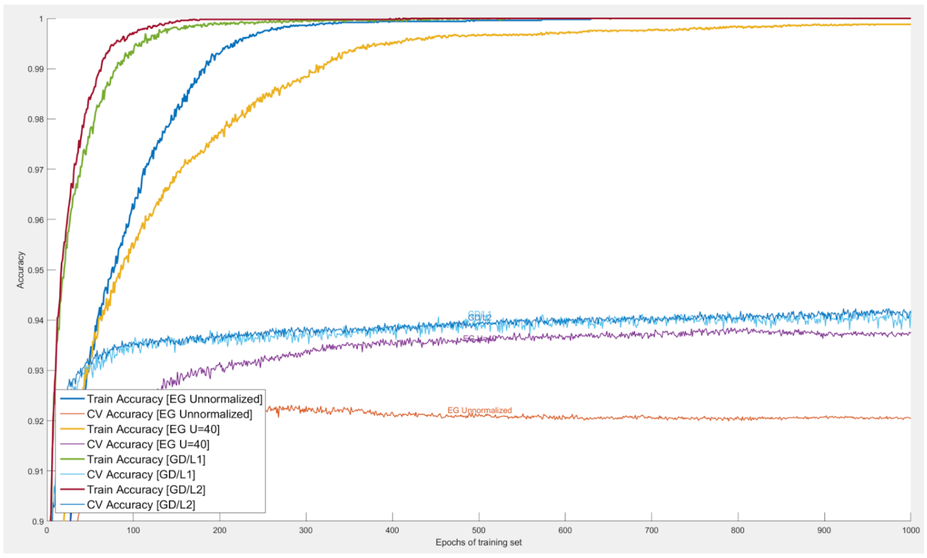


FIGURE 3.5: Training & testing accuracy on three networks (1) MNIST trained with EG$^{\pm}$ with U=40, (2) EG$^{\pm}$ unnormalized, and (3) SGD using L1 & L2 regularization with per-neuron normalization.

to note how similarly the distributions are.

3.4 Unnormalized EG

We analyzed EG$^{\pm}$ with normalization against EG without normalization. We found that EG can function without normalization, however it reduces it's effectiveness significantly. The following comparison trains on MNIST using a fully connected network with 1 hidden layer. We show a training set vs held out test set, with both training accuracy and test accuracy shown for EG$^{\pm}$ with normalization, and $U = 40$, EG$^{\pm}$ unnormalized, and SGD with L1 and L2 regularization. This example uses a subset of the full MNIST samples (a subset is used for computational efficiency), hence the accuracy is not expected to be state-of-the-art, this was done for computational efficiency.

EG$^{\pm}$ unnormalized performs a few percent worse in testing accuracy, and similarly for training accuracy. EG$^{\pm}$ unnormalized also trains more slowly.

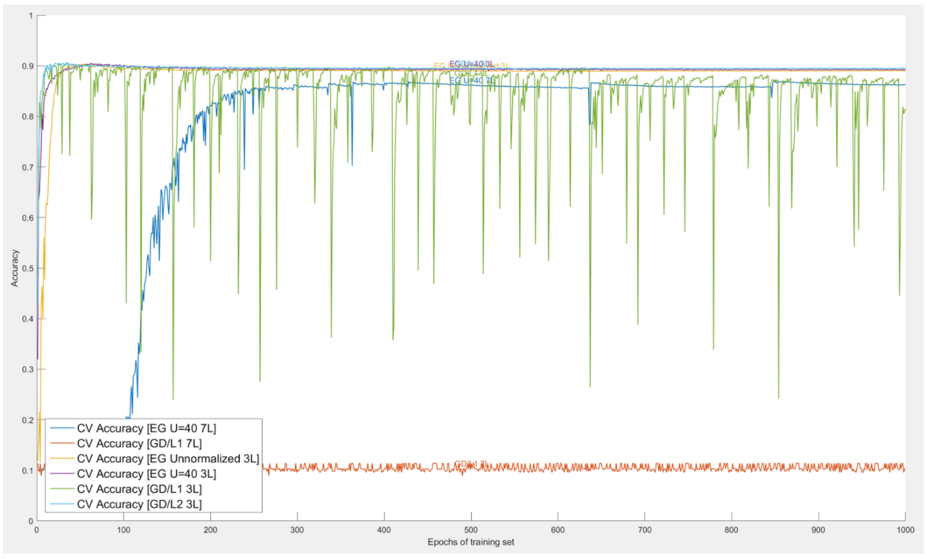


FIGURE 3.6: 6 networks compared, all trained on a 5k subset of MNIST: The first two are 7-layer fully connected networks EG$^{\pm}$ (U=40) and SGD with L1 regularization. The last 4 are 3-layer fully connected networks with EG$^{\pm}$ unnormalized, EG$^{\pm}$ (U=40), and SGD with L1 and L2 regularization applied. All with per-neuron normalization.

3.5 EG+/- with random noisy features

Notably EG$^{\pm}$ is expected to perform well when there are many noisy features because it promotes sparsity in the weights akin to L1 regularization. In this experimentation we've trained a fully connected feed forward neural net, with 3 layers, on the MNIST dataset, and we've added features which consist purely of zero-mean, unit-variance, Gaussian noise. Figures 3.6, and 3.7 show the results of the two experiments. In the first, Figure 3.6 we have added 784 of random Gaussian noise to the original 784 features of MNIST (28x28 images = 784 features). In the second, Figure 3.7, we add 10,000 features of random Gaussian noise to the original 784 features of MNIST. EG$^{\pm}$ has no problem training on the datasets with random features, seeming to ignore them. Whereas SGD optimization has more significant issues with the noisy dataset.

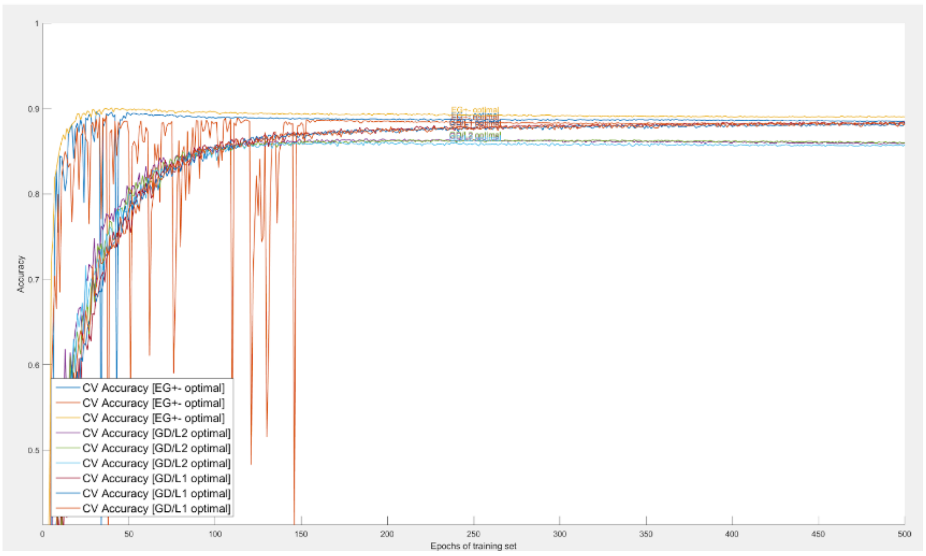


FIGURE 3.7: EG$^{\pm}$ and SGD with L1, and SGD with L2 regularization are trained 3 times with different random initialization states. Also all hyperparameters have been carefully optimized for each model to ensure the best possible test set accuracy. All with per-neuron normalization.

In Figure 3.6, focusing on the 3 layer networks we see that EG, both unnormalized and normalized perform quite stably, however SGD shows some instability in training with the noise.

In the second Figure 3.7 the superiority of EG$^{\pm}$ is much more pronounced. In this training we've carefully optimized the hyperparameters of EG and SGD to ensure both have the best possible results. EG$^{\pm}$ is clearly able to outperform SGD, regardless of L1 or L2 regularization used on SGD.

3.6 Comparing EG$^{\pm}$ with other optimization methods

In order to perform the most direct comparison EG$^{\pm}$ with other methods we have added EG$^{\pm}$ to the ConvnetJS framework which provides an implementation of a 2 layer fully connected network trained on the full MNIST dataset using the many optimization methods discussed in this thesis.

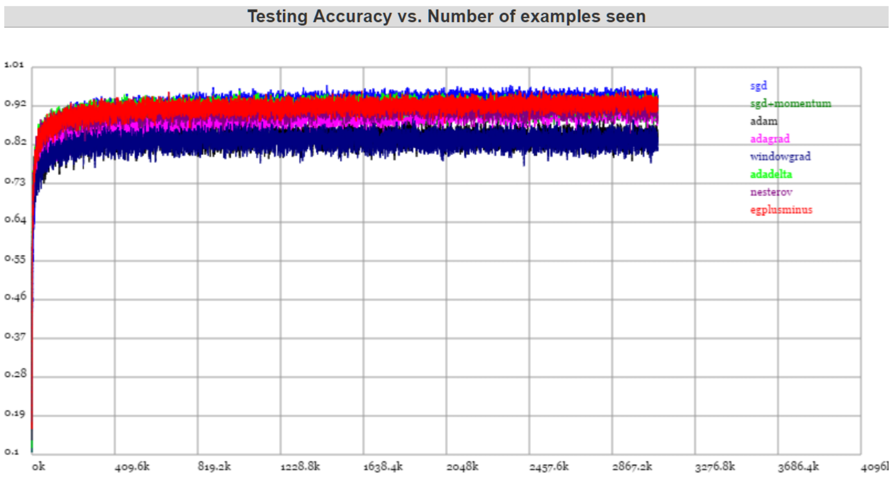


FIGURE 3.8: Running all trainers in ConvnetJS, including the added EG$^{\pm}$ trainer. EG$^{\pm}$ performs well in this head to head comparison, comparing to the top among trainers: SGD, SGD+Momentum, Adam, Adagrad, Windowgrad, Adadelta, and Nesterov.

In the results presented in Figure 3.8 EG$^{\pm}$ is trained on the same dataset, seeing 300,000 samples of the dataset during training. Accuracy is plotted on the test dataset. Of particular note, to keep the results as comparable as possible the same degree of L2 regularization on the cost function (softmax/cross entropy) was applied across all optimizers, including EG$^{\pm}$.

For EG$^{\pm}$ the learning rate was set to a small value, 0.00005, found by cross validation trial and error, and the EG$^{\pm}$ U scaling parameter was set to a value of 40. Note that the value of the learning rate is affected by the choice of loss functions and can't be compared directly between the two implementations of EG$^{\pm}$ used in this thesis.

The visual in Figure 3.8 is a little general, so another run using the same configuration was trained until 1M training samples had been iterated over, the results are in table 3.1. In this case EG$^{\pm}$ performed at the top of the group on a held out test set, achieving accuracy of 0.93125 at the end, slightly beating SGD at 0.92875, and Adadelta at 0.92125.

This result is unexpected for a few reasons. First, SGD wasn't expected to perform near the top of the group. Second, many of the more tuned algorithms such as Adam didn't perform as well as they have been observed to perform in other settings, Adam achieved a test set accuracy of only 0.8275, significantly worse than EG$^{\pm}$, SGD, or Adadelta.

This result for EG$^{\pm}$ contradicts some of the results obtained in other experiments, and the key difference that stands out is that L2 regularization was applied on the loss function for EG$^{\pm}$ in this experiment, whereas we steered away from applying regularization on the loss function for EG$^{\pm}$ in other experiments. In other experiments we opted to rely on the algorithm itself to perform its own form of regularization.

samples	SGD	SGD+Momentum	Adam	Adagrad	Windowgrad	Adadelta	Nesterov	EG$^{\pm}$
50k	81.3%	81.8%	73.4%	78.0%	76.9%	87.4%	84.4%	84.8%
100k	86.5%	87.9%	80.8%	82.3%	80.5%	90.8%	84.9%	87.5%
250k	89.3%	90.6%	80.9%	86.9%	84.1%	90.8%	89.1%	89.9%
500k	89.6%	88.4%	80.9%	85.4%	84.1%	90.5%	88.4%	90.3%
750k	90.6%	89.8%	81.6%	87.9%	83.3%	90.3%	89.6%	91.9%
1M	92.9%	90.8%	82.8%	87.9%	85.4%	92.1%	89.1%	**93.1%**

TABLE 3.1: Comparing trainers on MNIST, displaying test set accuracy over time. Rows are per number of samples trained on, columns are per optimization algorithm.

3.7 Applying adadelta's per-weight learning rate to EG$^{\pm}$

Based on the success we saw in Section 3.6, we decided to try to pull in some of the features that have worked in other gradient descent techniques to try to improve EG$^{\pm}$ further. A significant amount of work has gone into devising custom per-weight learning rates. We now experiment with utilizing Adadelta's (Zeiler, 2012) method of computing the learning rate per weight and apply it to EG$^{\pm}$.

Adadelta was introduced as an improvement over Adagrad's approach to computing a learning rate per weight. For a review of Adagrad see 1.2.4. Adagrad computed a per-weight learning rate by maintaining a sum of square gradients history of the gradient, allowing weights with a small gradient to take larger step sizes. Adadelta (see 1.2.5 improved upon Adagrad by maintaining a decaying history of both the weight values and gradients and using the ratio of the two to adjust the learning rate. Adadelta succeeds in not decaying the learning rate over time. Furthermore Adadelta completely replaces the learning rate parameter, whereas Adagrad still required a global learning rate parameter.

This effort takes the Adadelta method of computing the learning rates and uses that as the learning rate for EG$^{\pm}$. This has one immediate challenge in that the per-weight learning rates computed by the Adadelta method tend to stay around zero to one. Anything near one will be much too high of a learning rate for EG$^{\pm}$, causing numeric overflow (Infinity values), so we further scale the per-weight learning rate by a fixed global learning rate.

The vector of Adadelta per-weight learning rates $\boldsymbol{\eta}$ is computed via the following formulas:

$$\boldsymbol{G}_{t+1} = \rho \cdot \boldsymbol{G}_t + (1-\rho) \cdot \nabla_{\boldsymbol{w}} J(\boldsymbol{w})_t{}^2 \tag{3.1}$$

$$\boldsymbol{\eta_{t+1}} = \sqrt{\frac{\boldsymbol{X}_t + \epsilon}{\boldsymbol{G}_{t+1} + \epsilon}} \tag{3.2}$$

$$\boldsymbol{X}_{t+1} = \rho \boldsymbol{X}_t + (1-\rho)(\boldsymbol{w}_{t+1} - \boldsymbol{w}_t)^2 \tag{3.3}$$

$\boldsymbol{\eta}$	The per-weight learning rates to be used by EG$^\pm$ (in combination with the global learning rate scaling parameter)
ρ	A hyperparameter, usually fixed to a value above 0.9, we use 0.95.
$\boldsymbol{G}$	The gradient sum, this vector stores a decaying sum of past square gradients.
$\boldsymbol{X}$	this vector stores a decaying square-sum of changes in the weight values.
$\boldsymbol{w}$	the vector of neural network weights, in the case of EG$^\pm$ these are the weight value after taking the difference between positive and negative weights.
$\nabla_{\boldsymbol{w}} J$	the gradient of the weights with respect to the loss function. This is the gradient calculated by backprop.
ϵ	is a small value to avoid numerical issues.
t	The iteration step, t+1 is the updated iteration step, t is the previous iteration step.

It should also be noted that the $\boldsymbol{X}$ sums were initialized to 1.0 in this implementation. An issue with poor initialization was discussed in Section 1.2.5, and is particularly problematic with EG$^\pm$ because it causes an exploding weights issue (Infinity values aren't hard to come by in exponentiation).

The ultimate result of this effort performs at approximately the same level with EG$^\pm$, and it's notable that the global learning rate needed to be tuned carefully to achieve the results. We present the results of training EG$^\pm$, and EG$^\pm$ with Adadelta adaptive learning rates, and 7 other methods in table 3.2.

samples	SGD	SGD+Momentum	Adam	Adagrad	Windowgrad	Adadelta	Nesterov	EG$^\pm$	EG$^\pm$ Adaptive LR
885k	92.3%	90.8%	72.1%	89.3%	85.1%	92.1%	91.4%	92.9%	92.2%

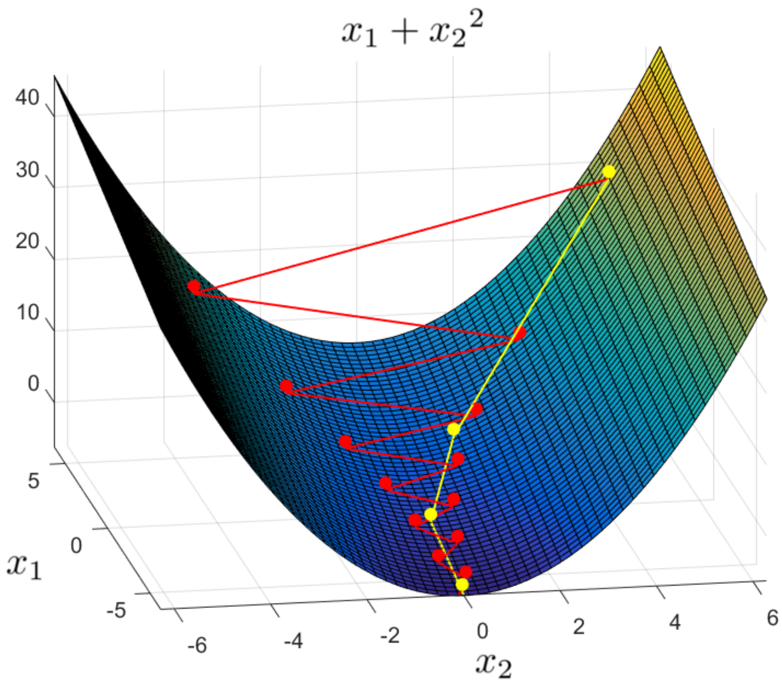


FIGURE 3.9: EG$^{\pm}$ in red, EG$^{\pm}$ with Adagrad adaptive learning rate in yellow (with "good" initialization, see 1.2.5.1).

TABLE 3.2: Comparing all gradient optimization methods on a 2 layer MNIST dataset, including EG$^{\pm}$ with adaptive learning rate from Adadelta, the adaptive EG$^{\pm}$ uses a global learning rate parameter of 0.02, U=40, and ρ=0.95.

We would also like to take a look at how EG$^{\pm}$ with adaptive learning rates performs on the 2D visualization presented for all of the other optimization methods. Figure 3.9 shows EG$^{\pm}$ updates vs. EG$^{\pm}$ with adaptive learning rates. In this 2D simplified example there certainly appears to be an improvement in the results, however in the practical example applying EG$^{\pm}$ to MNIST, it didn't make a notable difference.

3.8 Sum of square loss vs. cross entropy

During experimentation both square loss and cross entropy were used on MNIST datasets. Cross entropy is well known to perform better on classification tasks such as MNIST, however it was notable that when we used square loss, even though it's inferior to cross entropy for classification, EG$^{\pm}$ performed better than SGD in the case of square loss, but when cross entropy was applied SGD with L1 or L2 regularization on the loss function out performed EG without regularization on the loss function. This note is added for completeness sake, there was no further follow up regarding the choice of loss functions. All further results utilize cross entropy as a loss function.

3.9 EG+- compared vs. SGD/L1 regularization

In this experiment we hypothesize that EG$^{\pm}$ its self is performing a form of L1 regularization. We compare EG$^{\pm}$ with varying U scaling parameters from 20 to 70 to SGD with L1 regularization of the weights being added to the loss function, varying the amount of regularization across an appropriate range of values.

This network is trained on a 5k subsample of the MNIST dataset, and results are reported on a held out test set of 10k samples.

For the top 3 values of L1 regularization the results in Figure 3.10 show that SGD with L1 regularization out performed EG$^{\pm}$ with no regularization on the cost function, over a variety of U scaling parameters. This suggests that regularization on the cost function is a more significant advantage, in particular we find that regularizing EG$^{\pm}$ seems to be beneficial in Section 3.6.

3.10 Sharing weights applied to EG$^{\pm}$

Weight sharing is a concept drawn from extensive work in online learning. In the online learning context weight sharing is the process of adding some past history of the weight values to the current values, using a distribution such as uniform, decaying past, and other more complex techniques. These approaches

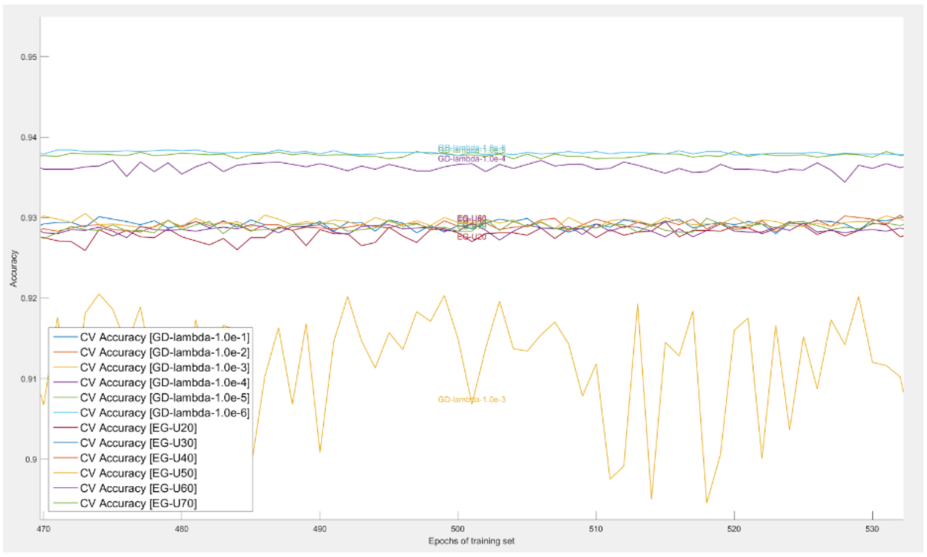


FIGURE 3.10: 6 networks trained with SGD and L1 regularization on a cross entropy loss function with varying lambda regularization parameters, vs. 6 networks trained with EG$^{\pm}$ given varying U scaling parameters and per-neuron normalization. SGD with L1 regularization out performs EG$^{\pm}$.

have the benefit in the online learning setting weights can return more quickly to a previously remembered state.

In the context of gradient descent optimization we have experimented with sharing a past average of weights to the current update step. The percentage of past average weight is a fixed hyperparameter, α.

The weight update step has the following addition to share the past average:

Algorithm 3: EG$^{\pm}$ sharing update

```
Data: Neural network weights, $\boldsymbol{\theta}$, past average $\boldsymbol{A}$ of the weights, and $\alpha$
      sharing parameter
Result: Updated weights $\boldsymbol{\theta}$ with sharing applied
for each weight: $\theta_i$, and past average $A_i$ do
    Update $\theta_i$ with EG$^{\pm}$ algorithm, unnormalized
    $\theta_i \leftarrow (1 - \alpha) \cdot \theta_i + \alpha \cdot A_i$
    Normalize as per the standard EG$^{\pm}$ process
end
```

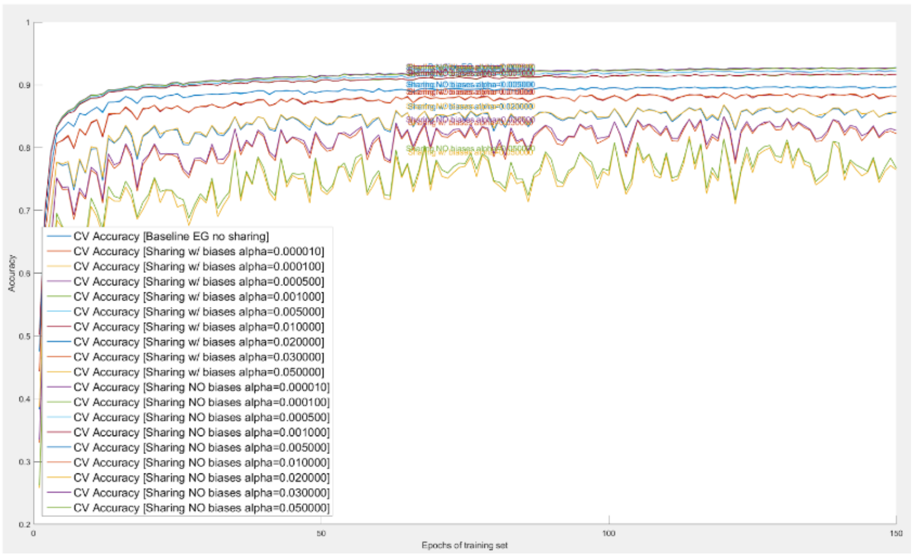


FIGURE 3.11: Results of weight sharing for increasing values of α, the more sharing was added, the worse the performance of the network.

In Figure 3.11 the results of sharing were clearly not beneficial to the network. In Figure 3.12 we can understand the likely cause. Sharing as it's implemented here was designed to bring weights back to a previous state. In the online learning setting this is a desirable property. However in the setting of gradient descent optimization, drawing the weights back to a previous state is not our objective. Figure 3.12 shows the movement of 20 randomly selected weights as they train. The values of those weights converges to a value, but does not jump around or return to a previous state as they might in the online learning setting.

Based on this experimentation, sharing as it's performed in the online learning setting isn't expected to be directly applicable to EG$^{\pm}$ in the gradient descent domain. However changes to the way sharing is applied might have some beneficial value still, Adagrad (1.2.4), Adadelta (1.2.5), and Adam (1.2.8) all have forms of sharing past averages that are interpreted as adaptive learning rates.

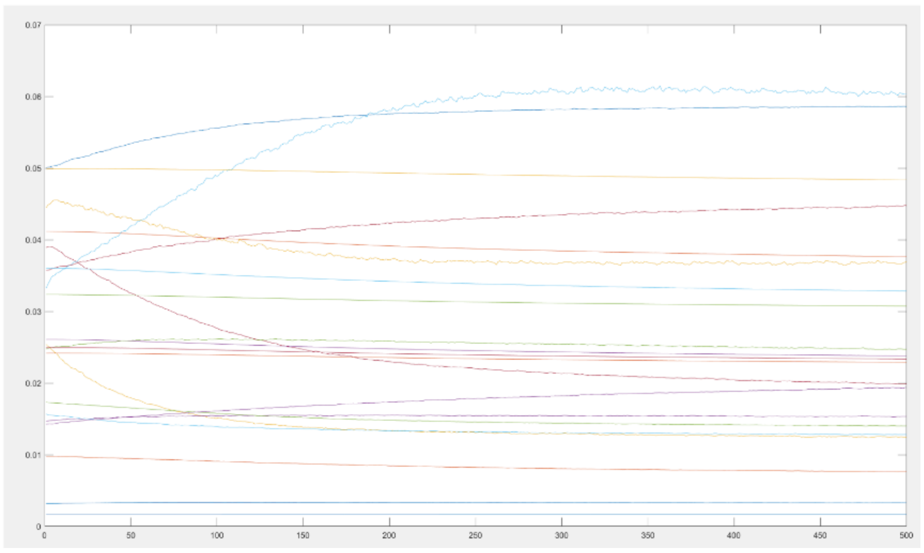


FIGURE 3.12: A plot of the movement of 20 randomly selected weights as EG$^{\pm}$ trains with per-neuron normalization. In the first few steps some weights train in one direction and then change, but after the initial few steps all weights simply converge to a value, never returning to a previous state.

3.11 Overfitting with SGD vs EG$^{\pm}$

We experiment here with overfitting by comparing EG without regularization against SGD with L2 regularization on networks with large numbers of weights. We use a 6 layer fully connected feed forward network. Fully connected neural networks tend to perform worse as you add more and more layers. This is due to a combination of factors including overfitting due to a large number of weights and the vanishing gradient problem. We utilize this architecture to test how EG$^{\pm}$ performs in these less than optimal conditions against SGD.

In Figure 3.13 the results confirm what we've seen in other experiments, that SGD with L2 regularization will out perform EG$^{\pm}$ with no regularization. Based on results presented in earlier sections we can surmise that adding L2 regularization to the loss function for EG$^{\pm}$ would have improved the results for EG$^{\pm}$.

One rather interesting result of note here is that we compared both square

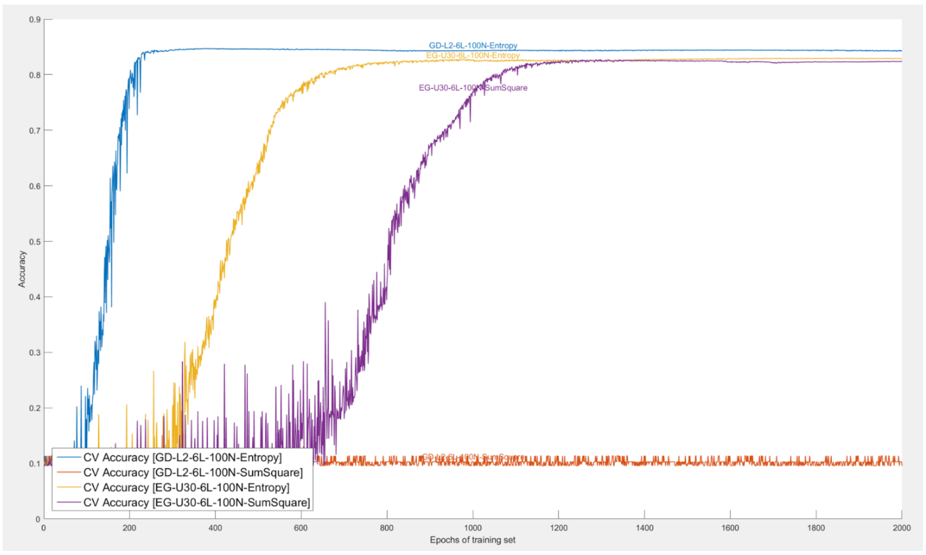


FIGURE 3.13: Four networks are trained using EG$^{\pm}$ and SGD using the MNIST dataset. Square loss is used for two (which is non optimal), and cross entropy is used for two. SGD with square loss fails to train, where EG$^{\pm}$ does, but with cross entropy (the correct loss function for classification), SGD with L2 regularization outperforms EG$^{\pm}$. All with per-neuron normalization.

loss to cross entropy loss in this case. It should be noted that square loss for a classification problem is distinctly non optimal. That non optimality aside, it is worth noting that EG$^{\pm}$ was able to train using square loss, whereas SGD failed to train with square loss. Whether this result holds for regression problems where square loss is the correct loss function was not tested, but this result does seem to suggest that EG$^{\pm}$ might have an advantage in regression problems.

3.12 Results of applying EG+- in a convolutional neural network

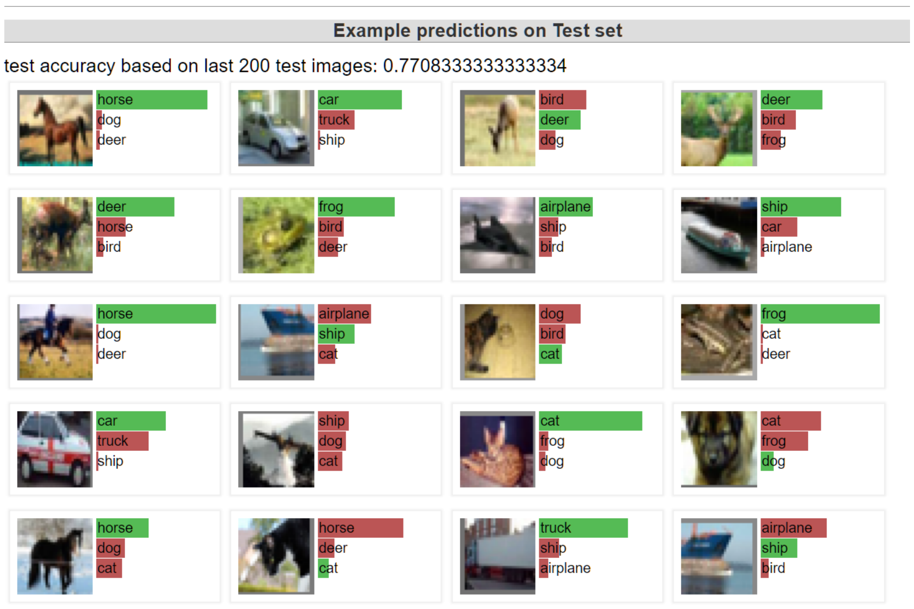
This section presents EG$^{\pm}$ in the context of a convolutional neural network as it's implemented in ConvnetJS. Convolutional neural networks are ubiquitous with image processing tasks such as classification. ConvnetJS implements a 3 layer convolutional neural network to do classification of the CIFAR10 dataset (Krizhevsky (2009)). The network uses ReLu activations and a softmax/cross entropy loss function.

We have trained a network for 200,000 iterations, with a batches of 4 images, and L2 regularization of 0.001 using both $EG^{\pm}$ and Nesterov. These networks produce a test set accuracy in the mid around 65% for both $EG^{\pm}$ and Nesterov, achieving higher accuracy with longer training (this platform is not efficient enough to run for significantly longer, with GPU support this network configuration can achieve accuracy close to 80%).

The results (see example 3.14) for training in this network are similar between $EG^{\pm}$ and Nesterov. At this point in training the Nesterov trainer achieves 61.2% accuracy on a random test set of the past 1000 test images, and the $EG^{\pm}$ trainer achieves 60.2% accuracy. The accuracy on test set has enough variance that these two results can be interpreted as quite similar.

In Section 3.3 we visualized the distribution of weights and biases that were generated from these two models. In this section we present further visuals generated by ConvnetJS with regards to the networks. We do not note significant differences between $EG^{\pm}$ and Nesterov trainers in this context. In particular the visualizations don't provide a definitive difference in terms of gradient activations, weight/bias distribution, or notable results regarding learning efficiency. More direct comparison were presented in Section 3.6. The visualizations here are available online at `https://goo.gl/68hIZb`.

The images in Figure 3.15 show just the first layer of the convolutional neural network. The input image is shown, with the activations of each pixel, and for the first convolutional network the activation values of each neuron are shown along with the backprop generated gradient value (where gray is 0, black represents a negative gradient, and white represents a positive gradient). Analyzing

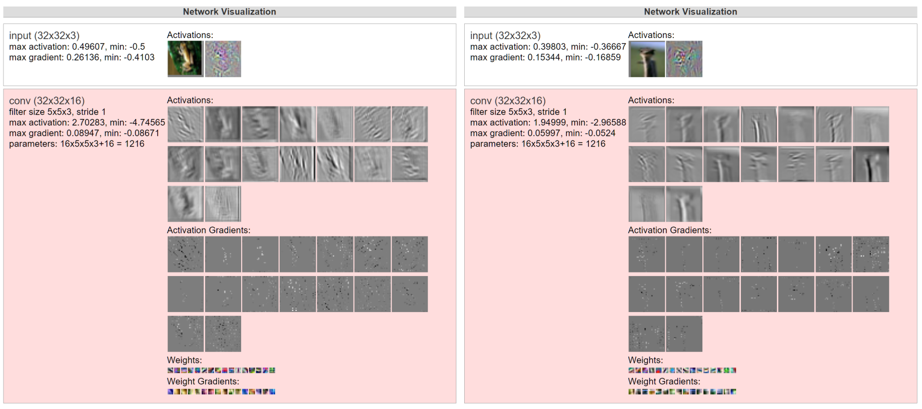


Example predictions on Test set

test accuracy based on last 200 test images: 0.7708333333333334

```
horse    car      bird      deer
dog      truck    deer      bird
deer     ship     dog       frog

deer     frog     airplane  ship
horse    bird     ship      car
bird     deer     bird      airplane

horse    airplane dog       frog
dog      ship     bird      cat
deer     cat      cat       deer

car      ship     cat       cat
truck    dog      frog      frog
ship     cat      dog       dog

horse    horse    truck     airplane
dog      deer     ship      ship
cat      cat      airplane  bird
```

FIGURE 3.14: Classification results of test images from CIFAR10 dataset using a 3 layer CNN trained with EG$^{\pm}$ for 200,000 iterations, a batch sizes of 4, and per-neuron normalization.



Network Visualization

```
input (32x32x3)
max activation: 0.49607, min: -0.5
max gradient: 0.26136, min: -0.4103

conv (32x32x16)
filter size 5x5x3, stride 1
max activation: 2.70283, min: -4.74565
max gradient: 0.08947, min: -0.08671
parameters: 16x5x5x3+16 = 1216
```

Network Visualization

```
input (32x32x3)
max activation: 0.39803, min: -0.36667
max gradient: 0.15344, min: -0.16859

conv (32x32x16)
filter size 5x5x3, stride 1
max activation: 1.94999, min: -2.96588
max gradient: 0.05997, min: -0.0524
parameters: 16x5x5x3+16 = 1216
```

FIGURE 3.15: Activation values (top), and the gradient values, whiteness represents positive gradients, blackness represents negative gradients, for each pixel in the first layer of the convolutional neural network. The left image was produced by EG$^{\pm}$, the right was produced with the Nesterov trainer.

these images for the two networks doesn't draw one to a conclusion that there is a major difference between the trainers, at least from the perspective of the activations and gradients being produced. It does seem that the problem set its self dictate the distribution of gradients and activations to a greater degree than the trainer itself does.

3.13 EG$^{\pm}$ on residual neural networks

In previous work EG$^{\pm}$ was tested on fully connected networks with the from-scratch Matlab implementation, and basic convolutional networks in Karpathy's javascript deep learning framework. The final reference implementation of EG$^{\pm}$ in Tensorflow makes it possible to test larger models that require GPUs to train.

We trained EG and various additive optimizers on a well respected residual network which has achieved state of the art on the cifar10 dataset recently. The version of the network used is derived from Tensorflow's research model He et al. (2016, 2015); Zagoruyko and Komodakis (2016). In this context we compare EG with its varying normalization forms to Vanilla SGD, SGD with Momentum, and Adam.

Dataset	EG Per Weight	EG Per Neuron	EG Per Variable	EG Full Graph	SGD	SGD+Momentum	Adam
10k Test samples	91.7%	81.3%	92.0%	88.0%	89.0%	89.0%	87.5%
Learning Rate	1e-3	1e-4	1e-4	1e-3	1e-4	1e-3	1e-4
Batch Size	128	256	256	128	128	128	256
U-Scaling	100	100	100	100	-	-	-
Momentum rate	-	-	-	-	-	0.98	-

TABLE 3.3: EG vs other optimizers compared using a 32 layer residual network on the cifar10 dataset.

Hyperparameter Search: The results in this experiment were obtained after running a hyperparameter search on each optimizer to determine the best settings for learning rate, batch size, and U-scaling parameter for each optimizer. The hyperparameter search followed a coordinate descent approach in which each attribute has a set of possible values assigned and each value is tested. The best result for each attribute is accepted after all values are tested and the next attribute is tested using the best result from each previous attribute. This process continues for at least 2 full iterations of all attributes. The process is validated empirically, but is not rigorous. The trade off for lack of rigor is speed and simplicity.

3.14 Adversarial examples

Black box adversarial samples, images in this case, are samples from the original dataset which are modified to fool the network into misclassifying the image. Such black box adversarial examples have been demonstrated to fool neural networks Xiao et al. (2018); Eykholt et al. (2017) even when access to the networks weights or ability to query is not provided. We have used such examples to test $EG^{\pm}$ in comparison to other additive gradient optimizers, see Figure 3.16. The experiment presented here is a 32 layer residual network trained on the cifar10 dataset using a variety of optimizers.

The results demonstrate that $EG^{\pm}$ performs significantly better than most other optimizers on the adversarial samples, with the notable exception of vanilla SGD which also performs quite well against them. The common feature of SGD and $EG^{\pm}$ is that neither uses momentum as part of their optimization algorithm. Therefore we postulate that momentum is the causal factor for the improved

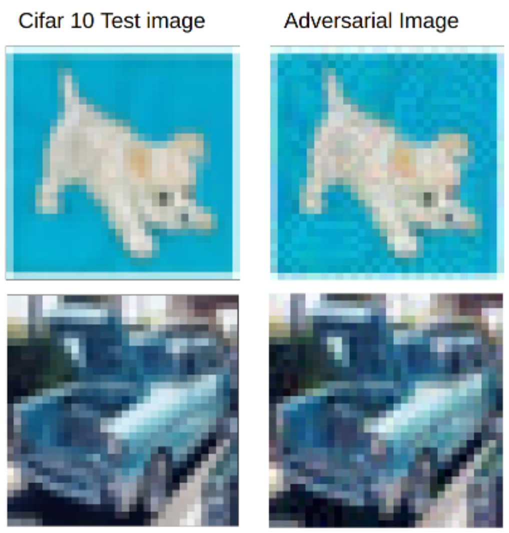


FIGURE 3.16: Visualization of two original cifar10 test images and the correspondingly altered adversarial image. Although they look quite similar, minor modifications can be noted. This is a black box adversarial attack, produced without access to the original model.

performance. Credit for this particular insight goes to Keller Jordan who first noticed that SGD performs differently than other additive optimizers.

Table 3.4 show the results of experiments training the 32 layer resnet with various optimizers and testing them against 1000 test-set samples from cifar10 which were originals or adversarial versions of the same image designed to fool the network using a blackbox attack.

Dataset	EG Per Weight	EG Per Neuron	EG Per Layer	EG Full Graph	SGD	SGD+Momentum	Adam
Test	91.7%	87.3%	93.1%	87.1%	85.4%	86.9%	87.0%
Adversarial	35.1%	54.5%	39.2%	48.2%	74.1%	54.7%	30.0%

TABLE 3.4: Comparison of EG optimizer with varying normalization method against other optimizers using 1000 test cifar10 samples and adversarial versions of the same images.

3.15 Conclusion

In this work we began by analyzing EG$^{\pm}$ and comparing it to vanilla SGD, and how EG$^{\pm}$ by its self would compare with SGD with L1 and/or L2 regularization. The regularized networks using SGD tended to out perform EG$^{\pm}$ except in the case where random noise was added as features. In these cases EG$^{\pm}$ out performed SGD even when SGD had L2 regularization.

In later work we found the EG$^{\pm}$ performed quite well against other popular gradient descent optimization algorithms. Notably when L2 regularization was included in the cost and gradient EG$^{\pm}$ seemed to perform best-in-class against other algorithms on a small 2 layer MNIST data set.

In the case of a more complex convolutional neural network using the CIFAR10 image classification dataset we found that EG$^{\pm}$ and Nesterov performed comparably, but due to the slow training times inherent in ConvnetJS, it wasn't possible to do exhaustive hyperparameter searches that are necessary to truly report results on these larger and more compute-intensive networks.

We attempted to combine per-weight learning rates of Adagrad, one of the better known gradient descent optimization algorithms to EG$^{\pm}$. While we show that this approach appears to work well in a 2D visualization example, it doesn't demonstrate a significant advantage when compared head to head with other trainers.

Finally we provide a reference implementation of EG$^{\pm}$ in tensorflow and demonstrate that it works comparably well to other gradient descent algorithms on modern ResNet architectures.

In summary, EG$^{\pm}$ appears to work well as a gradient descent algorithm, comparable with the best algorithms used in practice today. There are some special cases such as generated noise where it works better. EG$^{\pm}$ has some advantages over all but vanilla SGD on black box adversarial attacks.

Bibliography

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Z. Deng, C. Xu, Q. Cai, and P. Faraboschi. Reduced-precision memory value approximation for deep learning. 2015.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12 (Jul):2121–2159, 2011.

K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust physical-world attacks on deep learning models, 2017.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL http://proceedings.mlr.press/v9/glorot10a.html.

I. Goodfellow, Y. Bengio, and A. Courville. Deep learning. Book in preparation for MIT Press. URL http://www.deeplearningbook.org.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks, 2016.

G. Hinton. Rms prop. URL http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

e. a. Karpathy. Convnetjs - deep learning in your browser. Online. URL http://cs.stanford.edu/people/karpathy/convnetjs/.

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

J. Kivinen and M. K. Warmuth. Additive versus exponentiated gradient updates for linear prediction. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 209–218. ACM, 1995.

J. Kivinen and M. K. Warmuth. Exponentiated gradient versus gradient descent for linear predictors. *Information and Computation*, 132(1):1–63, 1997.

A. Krizhevsky. Learning multiple layers of features from tiny images, 2009.

J. Langford. Machine learning (theory) - exponentiated gradient, 08 2007. URL http://hunch.net/?p=286.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov. 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

Y. Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

M. A. Nielsen. Neural nnetwork and deep learning. Determination Press, 2015.

D. Parks. Experiemental neural network, matlab. Online, 2016. URL https://github.com/davidparks21/experimental_neural_network_matlab.

R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. *ICML (3)*, 28:1310–1318, 2013.

N. Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

T. Schaul, I. Antonoglou, and D. Silver. Unit tests for stochastic optimization. *arXiv preprint arXiv:1312.6055*, 2013.

N. Srebro and A. Shraibman. Rank, trace-norm and max-norm. In *International Conference on Computational Learning Theory*, pages 545–560. Springer, 2005.

N. Srinivasan, V. Ravichandran, K. Chan, J. Vidhya, S. Ramakirishnan, and S. Krishnan. Exponentiated backpropagation algorithm for multilayer feedforward neural networks. In *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP '02.*, volume 1, pages 327–331. IEEE, 2002.

N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

I. Sutskever. *Training recurrent neural networks*. PhD thesis, University of Toronto, 2013.

R. S. Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proc. 8th annual conf. cognitive science society*, pages 823–831, 1986.

C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song. Generating adversarial examples with adversarial networks, 2018.

S. Zagoruyko and N. Komodakis. Wide residual networks, 2016.

M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.