EE5121 Convex Optimization
Term Paper Report

R ADITHYA GOWTHAM
EE17B146

July 25, 2020

EXPLORING RECENT DEVELOPMENTS IN GRADIENT DESCENT METHODS

REFERENCE

Beyond SGD: Recent improvements of Gradient Descent Methods by Matthias Tschöpe, July 2019

CONTRIBUTIONS

Implemented the following algorithms in MatLab and visualized all algorithms using surface plots :

- Gradient Descent with fixed step-size
- Gradient Descent with backtracking line-search step-size optimizer
- Simple Momentum and Nesterov Momentum
- AdaGrad
- RMSProp with Momentum
- Adam and AdamW

INTRODUCTION

Optimization is extremely widespread nowadays, be it in subjects like Machine Learning, or activities like manufacturing, expenditure etc. As the society(mathematics and science in particular) gets more and more advanced, it becomes important to find effective optimization tools for dealing with certain problems, which makes finding a minimum or a maximum more efficient. Especially now since Deep Learning is booming, tools that are accurate and precise in optimizing functions, especially convex functions become more important as objective(or cost) functions are convex in a lot of scenarios.
More often than not, neural network processes contain differentiable functions. Making use of the properties of differentiation, recently there have been many developments in the field of descent techniques for convex functions. So, this paper explores the various improvements in gradient descent(GD) algorithms given in [Tsc19], the reference paper. The most widely used non-linear optimization algorithm nowadays is GD. But GD is not the state of the art optimization algorithm for ML. There are newer, more sophisticated but equally efficient optimization algorithms like the ones in the contributions section above. Assuming the all notations in convex optimization are self-explanatory, we proceed further (If not please refer to [Tsc19], Section 1.1 for notation) to the basic GD and its forms of implementation.

GRADIENT DESCENT

More often than not, the objectives that we tend to optimize are multi-dimensional. But for visualization sake, a 2-D input function and a 3-D surface are used for implementing the descent algorithms. So, the first step is to look for a descent direction to end up a local or global minimum. A vector $\mathbf{s} \in \mathbf{R}^{d_w}$ is a descent direction if $\nabla f(w)^T \cdot s < 0$ [BV14]. It has been proven in Lemma 2.1 of [Tsc19] that if $\nabla f(w) \neq 0$, then $s := -\nabla f(w)$ *is always a descent direction.* This can be proved easily by using the definition of a descent direction. Next, we try to prove that if $s := -\nabla f(w)$ and $\|s\|_2 = 1$, then s is always the direction of steepest descent. The proof is as below :

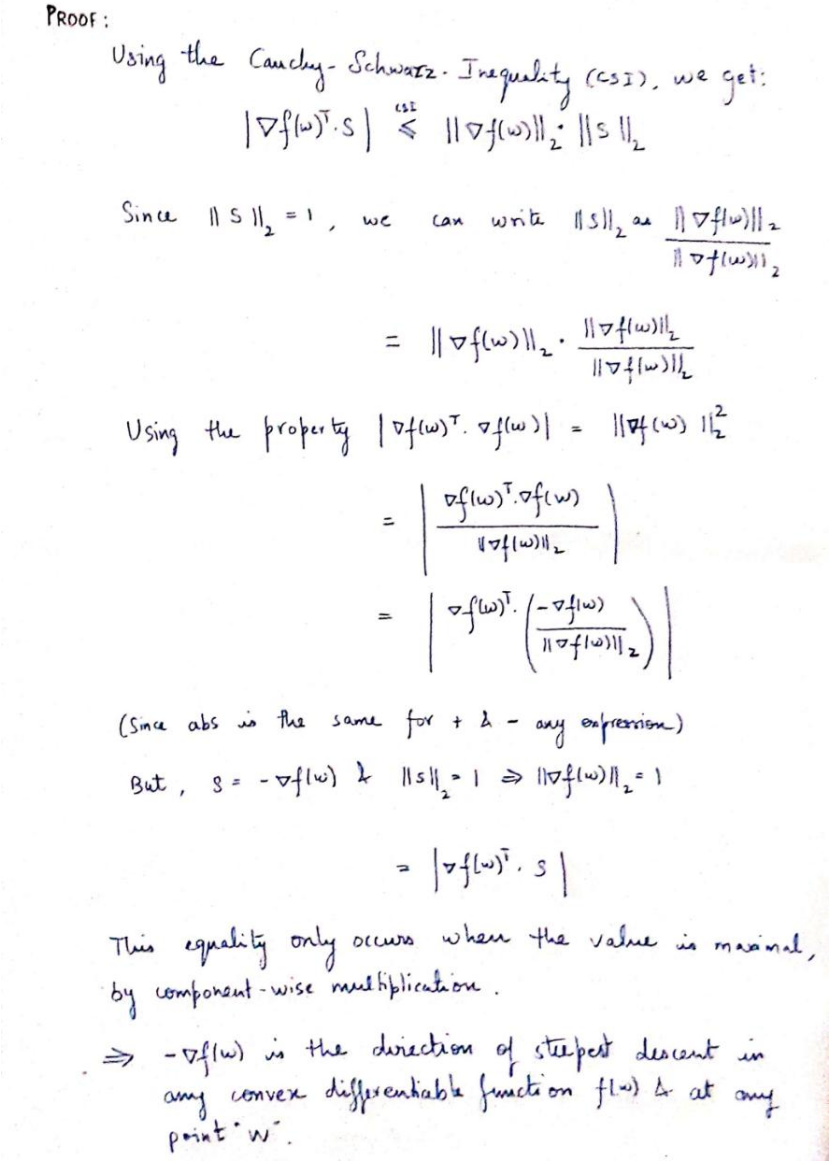


Figure 1: **Descent Direction proof**

From above, we can conclude that as long as x is not a stationary point, then choosing the negated gradient as the descent direction leads to the minimum most efficiently. So, the algorithm is termed Gradient(or Steepest) Descent. This algorithm is explored further in [Mez10] and [JZ13].

Algorithm 1: Gradient Descent$(f, x^{(0)}, \tau, \eta^{(0)})$

Input: Objective Function $f \in C^1(\mathbb{R}^{d_w}, \mathbb{R})$, start vector $x^{(0)}$, tolerance $\tau \geq 0$, step length $\eta^{(0)}$
Output: Almost stationary point $x^{(k)}$

```
k := 0
while ||∇f(x^(k))||_2 > τ do
    x^(k+1) = x^(k) − η^(k) · ∇f(x^(k))
    k := k + 1
return x^(k)
```

(The step where we update x is called the update step)
(Step-size is also called learning rate in Machine Learning.)

NOTE : The step-size for each iteration is different. This is because the step-size for each step of GD can be constant(this type is called Fixed-Step GD) or varying. There are many methods in [BV14] like Exact line search, Backtracking line search etc. used for optimizing step-sizes in GD.

There are two variations of Simple GD based on the amount of data used for the gradient update. These are :

- Stochastic Gradient Descent
- Mini-Batch Gradient Descent

Stochastic Gradient Descent (SGD)

In simple GD, we compute n derivatives in one iteration. But this is very time consuming [JZ13] especially if the dataset is very large(like 10 million entries). So for one update step, we need to loop through all 10 million values which makes the descent very slow. This is why the idea of SGD was introduced. In SGD, one randomly picked sample is used to calculate the gradient and then updates the value. This leads to larger variations in the descent direction, but it converges faster. Briefly, we minimize the expected risk instead of the empirical risk [Bot10]. The computation is n times faster than Simple GD [JZ13]. The following learning rate conditions are sufficient to ensure convergence [Bot10];[Goo+16]:

$$\sum_{k=1}^{\infty} \eta^{(k)} = \infty \qquad\qquad \sum_{k=1}^{\infty} (\eta^{(k)})^2 \leq \infty$$

Mini-Batch Gradient Descent (SGD)

Mini-Batch GD combines the advantages of GD and SGD by using neither one nor all samples but a small randomly selected subset $\tilde{X} \subseteq X$ of the dataset to calculate the gradient and perform the updates. The number $\tilde{n} := |\tilde{X}|$ of elements in $\tilde{X}$ is fixed. This allows the code to run fast and at the same time does not allow large variation in the descent steps.

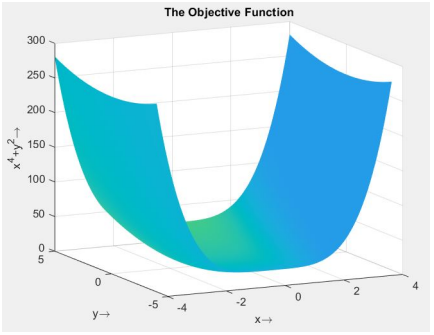
NOTE : If we use $\tilde{n} = n$, we get the GD algorithm and if we choose $n = 1$, we get the SGD algorithm.

THE OBJECTIVE FUNCTION USED FOR VISUALIZATIONS

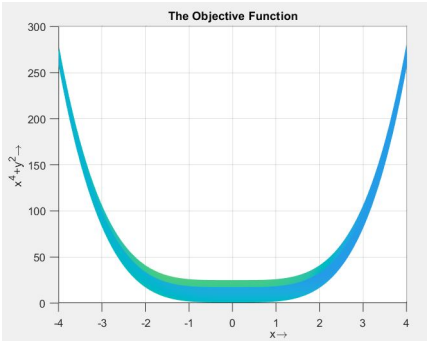
The objective function used for visualizing the descent algorithms in the Matlab scripts is :

$$f(x, y) = x^4 + y^2;$$

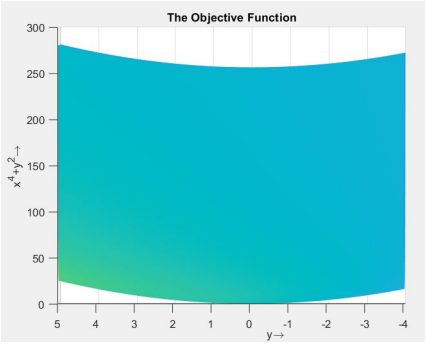
The surface plot of the objective as seen under different perspectives are shown below for better understanding :



(a) The Lateral view



(b) Front View



(c) Side View

Figure 2: **Surface plots of the objective function under different perspectives**

It is very lucid that the objective is convex and the global minimum is at $\boldsymbol{x}^* = (0, 0)$ with optimum value $f(\boldsymbol{x}^*) = 0$. We now try GD with fixed step-size on the objective and observe the results.

RESULTS

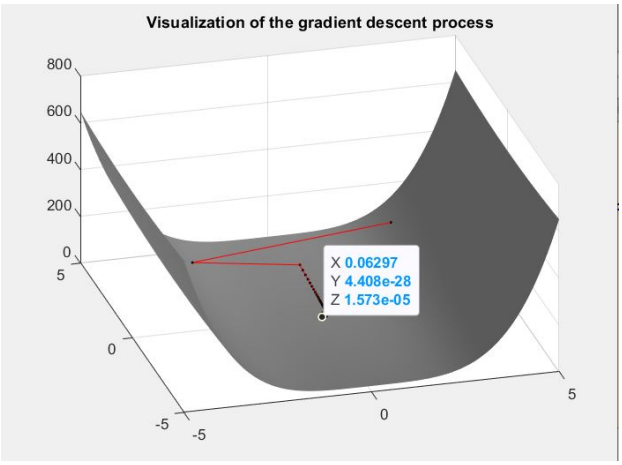


Figure 3: Visualization of GD with fixed step size

```
GRADIENT DESCENT WITH FIXED STEP SIZE
Hyperparameters: Step_Size = 5.000000e-02, Tolerance = 1.000000e-03

Number of steps taken to converge with fixed-step GD : 612
Convergence point x* = (6.297222e-02,4.407915e-28)
Optimized Objective at x* = 1.572519e-05
```

Figure 4: Convergence information of GD with fixed step size

Optimizing the objective using GD with backtracking line-search gives the following results :

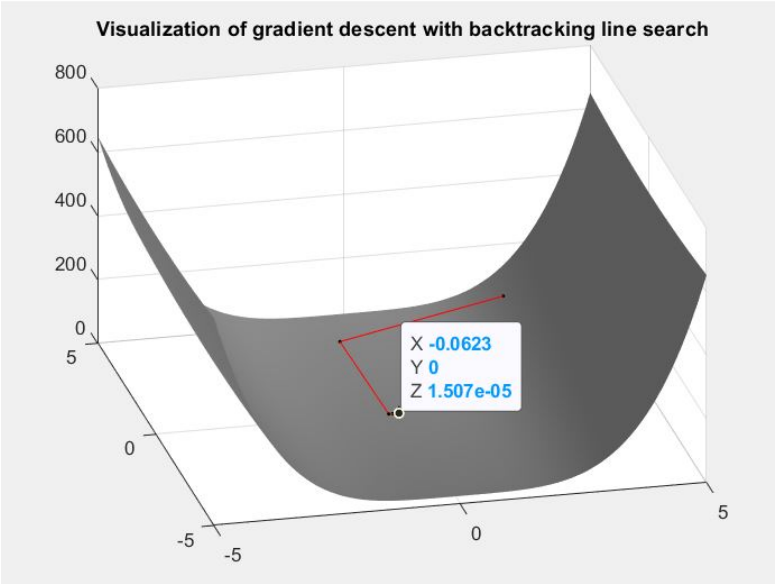


Figure 5: Visualization of GD with backtracking line-search

```
GRADIENT DESCENT WITH BACKTRACKING LINE_SEARCH
Hyperparameters: alpha = 1.000000e-01, beta = 5.000000e-01, Tolerance = 1.000000e-03

Number of steps taken to converge with backtracking line search : 31
Convergence point x* = (-6.230434e-02,0)
Optimized Objective at x* = 1.506861e-05
```

Figure 6: Convergence information of GD with backtracking line-search

---

MOMENTUM AND NESTEROV MOMENTUM

The earlier descent algorithms only use the first derivative and have no knowledge about the curvature of the objective function. This information is left unused. Imagine if we have to optimize an objective function with strong curvature, a large bulk of the information is present in the hessian $\nabla^2 f(w)$. This is computationally expensive with time complexity $O(d_w^2)$, where d_w denotes the dimensionality of $\vec{w}$. Note that there are algorithms like Newton's Method [BV14] which use the Hessian to optimize the objective. These are called "Second Order methods".

Momentum

The first mention of Momentum was from Polyak in 1964 in his paper [Pol64]. Momentum is an approach to estimate curvature information without actually computing the hessian [Goo+16]. The following figure from [Tsc19] compares Plain SGD with SGD with Momentum.

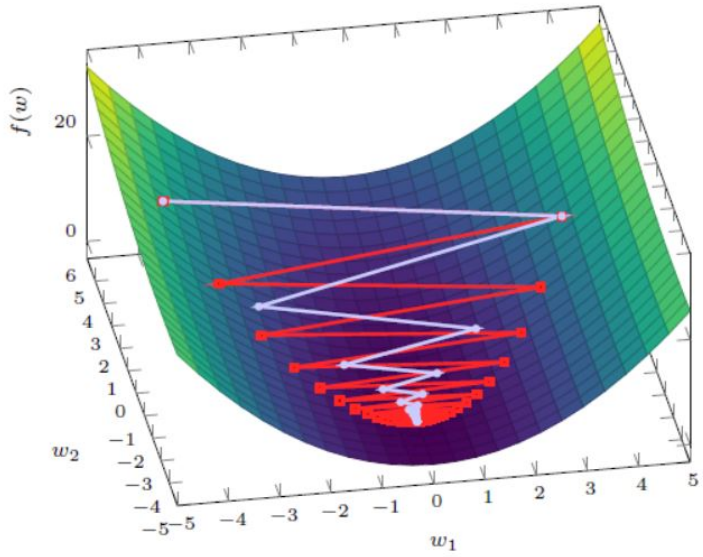


Figure 7: A surface of an objective function f(w) with the parameters w1 and w2 is shown. The red lines visualize the convergence trajectory of SGD without Momentum, the light blue lines visualize the convergence trajectory of SGD with Momentum. The 3D function f is quadratic and the Hessian of f is ill-conditioned. The ellipsoid surface of f is characteristic for an ill-conditioned Hessian. SGD needed 86 iterations to get the same accuracy as SGD with momentum after 30 iterations.

Please refer to [BBK08] and [MPS13] for detailed information on the condition number and how a hessian is ill or well-conditioned. To put it simply, If the hessian is ill-conditioned, then there exists at least one direction w_i, such that the slope in f(w) in direction w_i is much steeper than in all other directions [BBK08]. An example is given in Figure 1. It shows an objective function f with an ill-conditioned Hessian. The Figure also visualizes with the red lines, that SGD bounces more around the w_1 direction, and only makes small steps in w_2 direction. (A hessian is well-conditioned if it sloped relatively equally in all directions.)

Momentum can be better understood with a metaphoric real world situation. Consider that a ball is thrown in the descent direction(i.e. along -$\nabla f(w)$) at the start point of Figure(2). The path traced by the ball would first follow the descent direction, but as time goes on, the descent along w_2 direction increases over time (This is due to the component of normal force along w_2, i gravity acts vertically in the f(w) direction, putting it in physics terms) until the ball reaches the minimum. This idea when adopted into the GD algorithms gives rise to the Momentum approach. To realize this type of update, we add a new velocity vector $\vec{v} \in \mathbb{R}^{d_w}$ to our update rule[Goo+16].

Momentum updates can be visualized using the vector diagram below [Tsc19]:

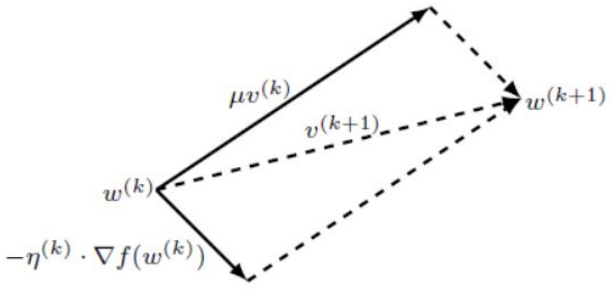


Figure 8: Visualization of the Momentum update rule.

The update equations now become :

$$v^{(k+1)} = \mu v^{(k)} - \eta^{(k)} \cdot \nabla f(w^{(k)})$$
$$w^{(k+1)} = w^{(k)} + v^{(k+1)}$$

where $v_{(k)}$ denotes the the value of $\vec{v}$ in the k-th iteration and $\mu \in [0, 1]$ is the momentum coefficient which determines how much momentum to carry over to the next iteration. We start with an initial velocity $v^{(0)}$ which is usually zero since the GD update takes care of the velocity over the next iterations (much like gravity accelerating a ball.

NOTE : If we set $\mu = 0$, the update rule is the same as GD/SGD.

RESULTS

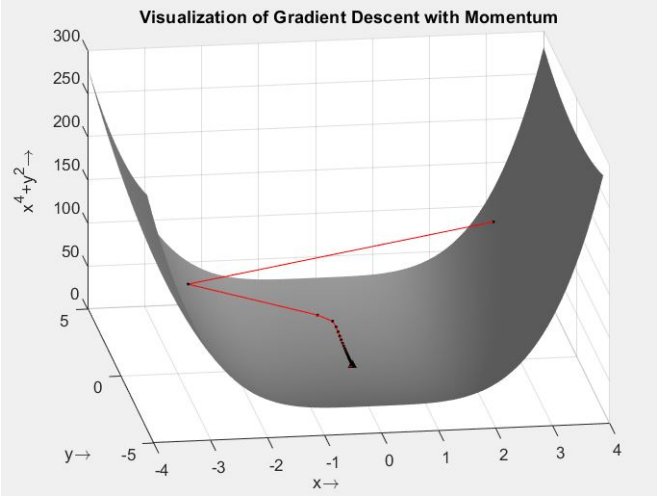


Figure 9: Visualization of Momentum

```
GRADIENT DESCENT WITH MOMENTUM
Hyperparameters: Mu = 1.000000e-01, Step_Size = 5.000000e-02, Tolerance = 1.000000e-03

Number of steps taken to converge with Momentum : 152
Convergence point x* = (6.297669e-02,5.856570e-08)
Optimized Objective at x* = 1.572966e-05
```

Figure 10: Convergence information of Momentum

Nesterov Momentum

In 1983, Yurii Nesterov proposed an improvement to momentum which computed the global minimum of a convex (but not necessarily strictly convex) function [Nes83] with a global convergence rate of $O(1/k^2)$ as compared to GD and SGD with a convergences rate of $O(1/k)$ for convex functions. In 2013, Sutskever et al. in their paper [Sut+13] came up with a new idea based on Yurii Nesterov's original idea, which was later called Nesterov Momentum [Goo+16].

Nesterov Momentum updates can be visualized using the vector diagram below [Tsc19] :
From the above diagram, the following update rule of Nesterov Momentum can be deduced :

$$v^{(k+1)} = \mu v^{(k)} - \eta^{(k)} \cdot \nabla f(w^{(k)} + \mu v^{(k)})$$
$$w^{(k+1)} = w^{(k)} + v^{(k+1)}$$

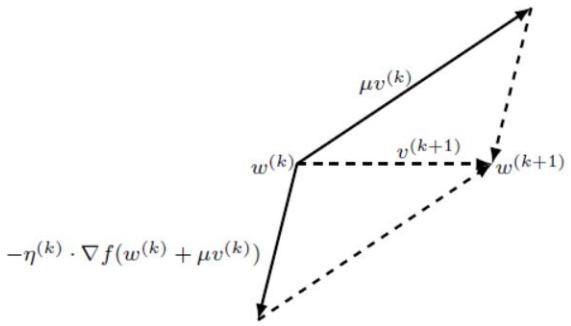


Figure 11: Visualization of the Nesterovs update rule. The velocity direction points too far in the upper east direction. Since Nesterov uses for the gradient the point $w^{(k)} + \mu v^{(k)}$ instead of $w^{(k)}$, this error is not as high as in Momentum.

RESULTS

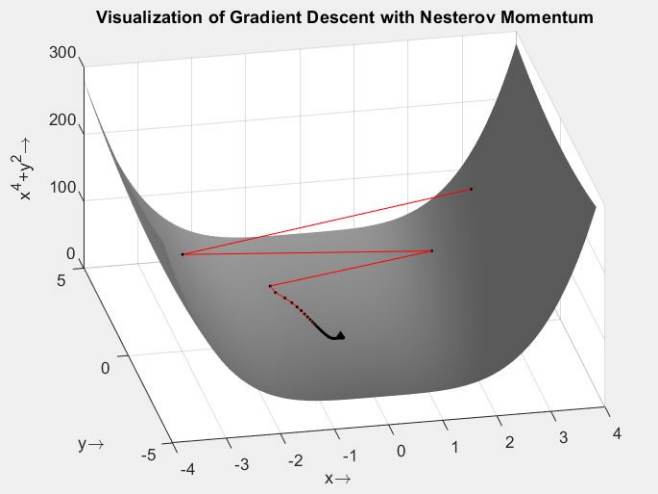


Figure 12: Visualization of Nesterov Momentum

```
GRADIENT DESCENT WITH NESTEROV MOMENTUM
Hyperparameters: Mu = 1.000000e-01, Step_Size = 5.000000e-02, Tolerance = 1.000000e-03

Number of steps taken to converge with Nesterov Momentum : 565
Convergence point x* = (-6.298715e-02,5.188129e-29)
Optimized Objective at x* = 1.574011e-05
```

Figure 13: Convergence information of Nesterov Momentum

The difference between Momentum and Nesterov Momentum is quite evident in the fact that the gradient is calculated at $w^{(k)} + \mu v^{(k)}$ instead of $w^{(k)}$. The idea is that the correction term $\mu v^{(k)}$ allows for more robust updates as explained in [Sut+13]. This makes Nesterov faster than Momentum but in the case of non-convex functions and especially for stochastic gradients with large variations in the gradient, Nesterov is not faster than Momentum [Sut+13][Goo+16].

ADAGRAD

The *AdaGrad* (short for *Adaptive Gradient*) Optimization Algorithm was introduced in 2011 by Duchi et al. in their paper [DHS10]. To understand AdaGrad better, we write the GD update equation in an alternate format :

$$\begin{aligned} w^{(k+1)} &:= w^{(k)} - \eta^{(k)} \cdot f(w^{(k)}) \\ &:= w^{(k)} - \eta^{(k)} \cdot \mathbf{1}_{d_w} \odot f(w^{(k)}) \end{aligned}$$

where $\mathbf{1}_{d_w}$ is the vector of d_w ones. We can see that GD or SGD use the same learning rate for all directions. The idea of AdaGrad is to use different learning rates for different directional gradients. The update rule for AdaGrad is :

$$w^{(k+1)} := w^{(k)} - \eta \cdot diag(G^{(k)})^{-1/2} f(w^{(k)})$$

where $G^{(k)}$ is defined as :

$$G^{(k)} = \sum_{j=1}^{k} (\nabla f(w^{(k)})) \otimes (\nabla f(w^{(k)}))^T$$

Please refer to [DHS10] for the derivations. Note that the mathematically precise version of AdaGrad computes $G^{(k)-1/2}$ which is NP-Hard for higher dimensional datasets[DHS10]. To mitigate this problem, Duchi et al. used $diag(G^{(k)})^{-1/2}$ instead, i.e only the entries from the main diagonal are used (these can be computed in $O(d_m)time$.

The learning rates $\boldsymbol{\eta}^{(k)}$; $k = 1...d_m$ are:

$$\boldsymbol{\eta}^{(k)} := \left(\frac{1}{\delta + \sqrt{\sum_{j=1}^{k} ((w^{(j)})_1)^2}}, ..., \frac{1}{\delta + \sqrt{\sum_{j=1}^{k} ((w^{(j)})_{d_w})^2}} \right)^T$$

where $\delta \sim 10^{-7}$ is added to the denominator to prevent divide-by-zero errors[BL17]. The update equation for AdaGrad can be rewritten as :

$$w^{(k+1)} := w^{(k)} - \eta \cdot \boldsymbol{\eta}^{(k)} \otimes f(w^{(k)})$$

In an algorithmic perspective, we sum of all previous gradients in an update variable $r^{(k+1)}$ so that future updates can be calculated instantly instead of summing up all gradients again. Also, if a gradient increases sharply in a particular direction (or the condition number in that direction is large) in a particular iteration k, and since the AdaGrad's adaptive step-sizes are inversely proportional to the directional gradients, the algorithm decreases adaptive learning in that direction much faster than other directions. **NOTE :** The convergence rate of AdaGrad in the case of non-convex functions in stochastic settings in $O(1/\sqrt{k})$ [WWB18]. In the case of batch settings and an optimal learning rate, the rate reduces to $O(1/k)$.

Algorithm 2: *AdaGrad*$(f, x^{(0)}, \tau, \eta)$

Input: Objective Function $f \in C^1(\mathbb{R}^{d_w}, \mathbb{R})$, start vector $\mathrm{x}^{(0)}, tolerance\tau \geq 0, steplength\eta$.
Output: Almost stationary point $x^{(k)}$

$\mathrm{k} := 0$
$\mathrm{r}^{(0)} := 0$
while $\left\|\nabla f(x^{(k)})\right\|_2 > \tau$ **do**
  $r^{(k+1)} = r^{(k)} + \nabla f(x^{(k)}) \odot \nabla f(x^{(k)})$
  $\boldsymbol{\eta}^{(k)} := \left(\frac{1}{\delta + \sqrt{r_1^{(k+1)}}}, ..., \frac{1}{\delta + \sqrt{r_1^{(k+1)}}} \right)^T$
  $x^{(k+1)} = x^{(k)} - \eta \cdot \boldsymbol{\eta}^{(k)} \odot \nabla f(x^{(k)})$
  $k := k + 1$
return $x^{(k)}$

RESULTS

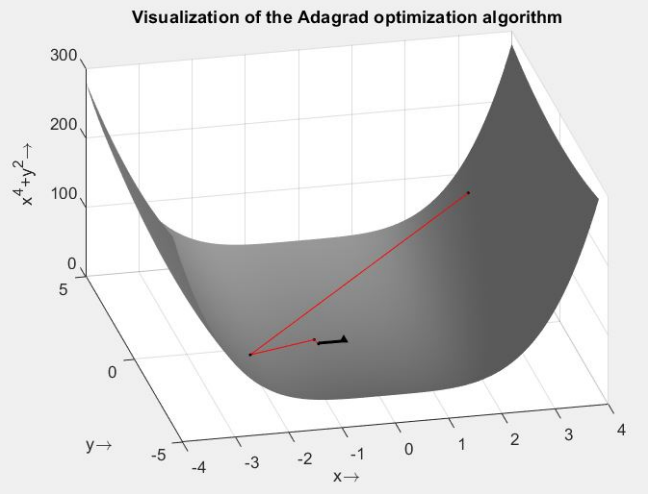


Figure 14: Visualization of AdaGrad

```
ADAGRAD
Hyperparameters: Step_Size = 5, Tolerance = 1.000000e-03, Delta = 1.000000e-05

Number of steps taken to converge with Adagrad : 701
Convergence point x* = (-6.299232e-02,0)
Optimized Objective at x* = 1.574528e-05
```

Figure 15: Convergence information of AdaGrad

RMSPROP

RMSProp stands for *Root Mean Square Propagation* and was introduced in 2012 by Hinton et al. in their paper [Sut+12] after explaining the disadvantages of Resilient Propagation. Recalling the earlier section, AdaGrad is very fast on convex functions. But in the case of non-convex functions, especially ones with large variations in gradient, AdaGrad reduces the adaptive learning rates very fast for steep decent directions than for others. This can lead to small learning rates before reaching a locally convex neighborhood which makes the algorithm converge much slower[Goo+16]. RMSProp uses an additional factor, $\rho \in \mathbf{R}$ along with adaptive learning rates to exponentially decrease the effect of past learning rates in the current update. Factoring in all these, the update rule for RMSProp is :

$$r^{(k+1)} := \rho^{(k)} + (1-\rho)\nabla f(x^{(k)}) \odot \nabla f(x^{(k)})$$

$$w^{(k+1)} := w^{(k)} - \eta \cdot \boldsymbol{\eta}^{(k)} \odot f(w^{(k)})$$

where $\boldsymbol{\eta}^{(k)}$ is the same as in AdaGrad. The authors of [Sut+12] and others use $\rho = 0.9$. The convergence guarantee for RMSProp was provided by Basu et al. in 2018 in their paper [Bas+18]. In practice, RMSProp is used along with Momentum/Nesterov.

(PTO)

Algorithm 3: *RMSProp_With_Momentum*$(f, x^{(0)}, \tau, \eta)$

Input: Objective Function $f \in C^1(\mathbb{R}^{d_w}, \mathbb{R})$, start vector $x^{(0)}$, tolerance $\tau \geq 0$, step length η, initial velocity $v^{(0)}$, momentum coefficient μ, decay rate ρ.

Output: Almost stationary point $x^{(k)}$

k := 0

$r^{(0)} := 0$

while $\left\|\nabla f(x^{(k)})\right\|_2 > \tau$ **do**

  $r^{(k+1)} = \rho r^{(k)} + (1-\rho)\nabla f(x^{(k)}) \odot \nabla f(x^{(k)})$

  $\boldsymbol{\eta}^{(k)} := \left(\frac{1}{\delta + \sqrt{r_1^{(k+1)}}}, ..., \frac{1}{\delta + \sqrt{r_1^{(k+1)}}} \right)^T$

  $v^{(k+1)} = v^{(k)} - \eta \cdot \boldsymbol{\eta}^{(k)} \odot \nabla f(w^{(k)})$

  $x^{(k+1)} = x^{(k)} + v^{(k+1)}$

  $k := k + 1$

return $x^{(k)}$

RESULTS

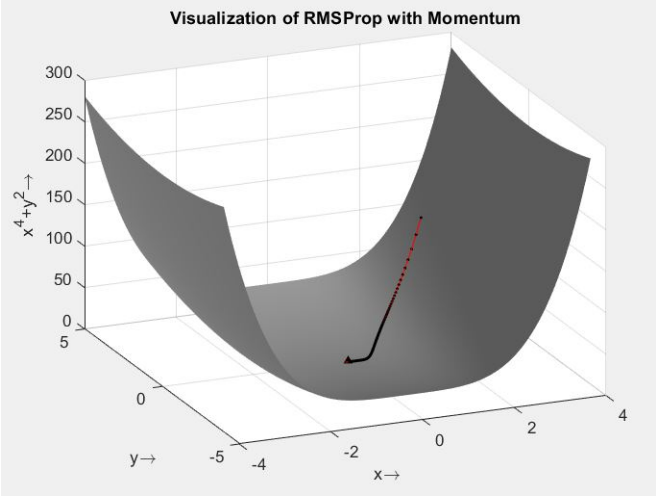


Figure 16: Visualization of RMSProp(with momentum)

ADAM

Adam stands for Adaptive Moment Estimation, proposed in 2014 by Kingma and Ba in their paper [KB14]. Adam shares a lot of similarities with the RMSProp with Momentum Algorithm earlier. But there are little but significant differences between the two. Adam uses first and second-order momentum vectors, where the first-order momentum vector is calculated as :

$$m^{(k+1)} = \beta_1 m^{(k)} + (1-\beta_1)\nabla f(x^{(k)})$$

This exponential smoothing (or weighted moving average [WJ03]) limits the influence of the new gradient as compared to RMSProp with Momentum. The second order momentum is the same $r^{(k+1)} \in \mathbf{R}^{d_w}$ that is used in RMSProp. Adam also uses bias correction for the first and second order moments that prevent biased momentum errors, which lead to large estimation errors[Goo+16]. The first order momentum is scaled by $\frac{1}{1-\beta_1^{k+1}}$ and the second order one by $\frac{1}{1-\beta_2^{k+1}}$ .This is also absent in RMSProp.

```
RMSPROP WITH MOMENTUM
Hyperparameters: Mu = 3.000000e-01, Step_Size = 5.000000e-02, Tolerance = 1.000000e-03,
                 Decay Rate = 9.000000e-01, Delta = 1.000000e-07

Number of steps taken to converge with RMSProp : 152
Convergence point x* = (6.260192e-02,9.517226e-07)
Optimized Objective at x* = 1.535856e-05
```

Figure 17: Convergence information of RMSProp(with momentum)

A **regret function** can be used to check the quality of an algorithm, defined as :

$$R(\tau) := \sum_{k=1}^{\tau} \Big(f(w^{(k)}) - f(w^*) \Big)$$

where τ is the iteration at which the algorithm goes below the tolerance and w^* is the global minimum (usually calculated manually). Kingma and Ba proved an average regret of $O(\frac{1}{\tau})$ for convex functions [KB14]. In 2018, Chen et al. proved an average convergence rate of $O(\frac{log(\tau)}{\tau})$ for non-convex functions.

Algorithm 4: $ADAM(f, x^{(0)}, \tau, \eta)$

Input: Objective Function $f \in C^1(\mathbb{R}^{d_w}, \mathbb{R})$, start vector $x^{(0)}$, tolerance $\tau \geq 0$, step length η, first order decay rate β_1, second order decay rate β_2.

Output: Almost stationary point $x^{(k)}$

$\mathrm{k} := 0$

$\mathrm{v}^{(0)} := \mathbf{0}_{d_w}$

$\mathrm{r}^{(0)} := \mathbf{0}_{d_w}$

while $\left\|\nabla f(x^{(k)})\right\|_2 > \tau$ **do**

$v^{(k+1)} := \beta_1 v^{(k)} + (1-\beta_1)\nabla f(x^{(k)})$

$r^{(k+1)} := \beta_2 r^{(k)} + (1-\beta_2)\nabla f(x^{(k)}) \odot \nabla f(x^{(k)})$

$\hat{v}^{(k+1)} := \frac{1}{1-\beta_1^{k+1}} v^{(k+1)}$

$\hat{r}^{(k+1)} := \frac{1}{1-\beta_2^{k+1}} r^{(k+1)}$

$x^{(k+1)} := x^{(k)} - \eta \left(\frac{\hat{v}_1^{(k+1)}}{\delta + \sqrt{\hat{r}_1^{(k+1)}}}, ..., \frac{\hat{v}_{d_w}^{(k+1)}}{\delta + \sqrt{\hat{r}_{d_w}^{(k+1)}}} \right)^T$

$k := k + 1$

return $x^{(k)}$

RESULTS

```
ADAM
Hyperparameters: Step_Size = 1.000000e-01, Tolerance = 1.000000e-03, Beta1 = 5.000000e-01,
                 Beta2 = 8.000000e-01, Decay Rate = 1.000000e-07, Delta =
Number of steps taken to converge with Adam : 151
Convergence point x* = (3.615879e-03,3.208608e-04)
Optimized Objective at x* = 1.031226e-07
```

Figure 18: Convergence information of Adam

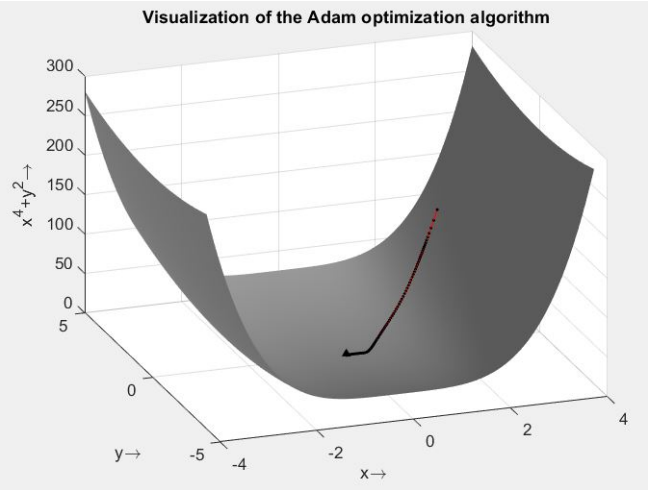


Figure 19: Visualization of Adam

AdamW

In many cases, especially in machine learning, many objective functions are heavily prone to overfitting. To prevent this, many types of regularization techniques are used, some of which include *Weight Decay, L_1 Regularization*(also called LASSO - Least Absolute Shrinkage and Selection Operator), or *L_2(or Ridge) Regularization* are used [Ng04]. The methods are defined as :

$$w^{(k+1)} := (1-\phi)w^{(k)} - \eta^{(k)} \cdot \nabla f(w^{(k)}) \qquad \text{(Weight Decay)}$$

$$f_{L_1}(w^{(k+1)}) := f(w^{(k+1)}) + \phi^{'} \left\| w^{(k)} \right\|_1 \qquad (L_1 \text{ Regularization})$$

$$f_{L_2}(w^{(k+1)}) := f(w^{(k+1)}) + \frac{\phi^{''}}{2} \left\| w^{(k)} \right\|_2^2 \qquad (L_2 \text{ Regularization})$$

NOTE : Weight Decay and L_2 regularization behave in the same way for SGD if $\phi^{''} = \frac{\phi}{\eta}$, where η is an additional hyperparameter. The proof is as follows :

Proof :
SGD without weight decay has the following update rule with L2 regularization :

$$w^{(t+1)} := w^{(t)} - \alpha \cdot \nabla f_(L_2)(w^{(t)}) = w^{(t)} - \alpha \cdot \nabla f(w^{(t)}) - \alpha\lambda^{'} w^{(t)}$$

SGD with weight decay (not L_2 regularization) has the following update rule :

$$w^{(t+1)} := (1-\lambda)w^{(t)} - \alpha \cdot \nabla f(w^{(t)})$$

These iterates are identical if we set $\lambda^{'} = \frac{\lambda}{\alpha}$.

However, in 2019 Ilya Loshchilov and Frank Hutter proved that L_2 regularization and Weight Decay behave differently for algorithms with adaptive learning rate. So, they decoupled η and ϕ and proposed two new algorithms SGDW and AdamW [LH19]. To Explore the difference between L_2 Regularization and Weight Decay for optimization algorithms with adaptive learning rate, we first derive the gradient of the L_2 regularized term.

$$\nabla \frac{\phi^{''}}{2} \left\| w^{(k)} \right\|_2^2 := 2\frac{\phi^{''}}{2} w^{(k)} = \phi^{''} w^{(k)}$$

$$\implies \nabla f_{L_2}(w^{(k+1)}) := \nabla f(w^{(k+1)}) + \phi^{''} w^{(k)}$$

Algorithm 5: Adam with L_2 Regularization and Adam with weight decay (AdamW)

Input: Objective Function $f \in C^1(\mathbb{R}^{d_w}, \mathbb{R})$, start vector $x^{(0)}$, tolerance $\tau \geq 0$, step length η, first order decay rate β_1, second order decay rate β_2, L_2 Regularization factor ϕ'', weight decay factor ϕ.

Output: Almost stationary point $x^{(k)}$

k := 0

$v^{(0)} := \mathbf{0}_{d_w}$

$r^{(0)} := \mathbf{0}_{d_w}$

while $\left\|\nabla f(x^{(k)})\right\|_2 > \tau$ **do**

$v^{(k+1)} := \beta_1 v^{(k)} + (1-\beta_1)(\nabla f(x^{(k)}) + \phi'' w^{(k)})$

$r^{(k+1)} := \beta_2 r^{(k)} + (1-\beta_2)(\nabla f(x^{(k)}) + \phi'' w^{(k)}) \odot (\nabla f(x^{(k)}) + \phi'' w^{(k)})$

$\hat{v}^{(k+1)} := \frac{1}{1-\beta_1^{k+1}} v^{(k+1)}$

$\hat{r}^{(k+1)} := \frac{1}{1-\beta_2^{k+1}} r^{(k+1)}$

$x^{(k+1)} := x^{(k)} - \eta \left(\frac{\hat{v}_1^{(k+1)}}{\delta + \sqrt{\hat{r}_1^{(k+1)}}}, \ldots, \frac{\hat{v}_{d_w}^{(k+1)}}{\delta + \sqrt{\hat{r}_{d_w}^{(k+1)}}} \right)^T - \eta\, \phi w^{(k)}$

$k := k + 1$

return $x^{(k)}$

We then look at the algorithm for Adam with L_2 regularization and Adam with Weight Decay(AdamW). Notice that the $L2$ Regularization in Adam affects both the gradient of the objective and the gradient of the regularization term in contrast to weight decay which affects only the gradient of the objective function. Also, note that the computation of the adaptive learning rate is independent of the regularization term $\phi w^{(k)}$ in AdamW [LH19].

RESULTS

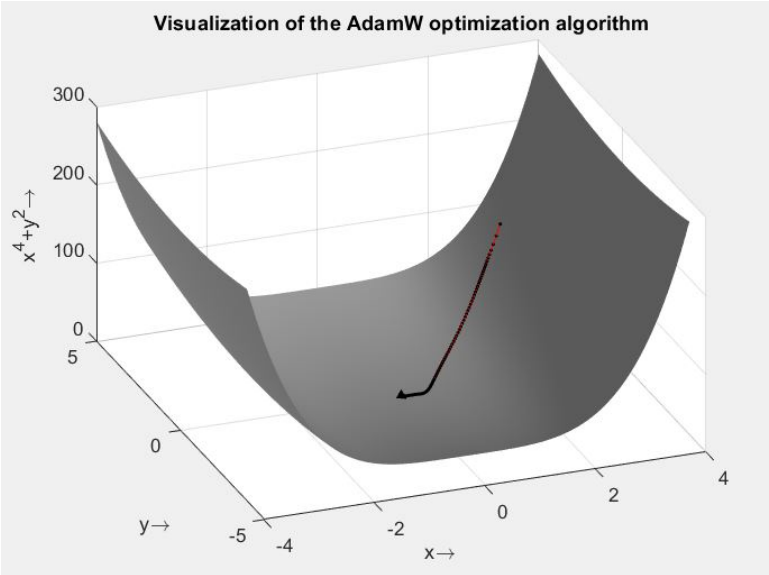


Figure 20: Visualization of AdamW

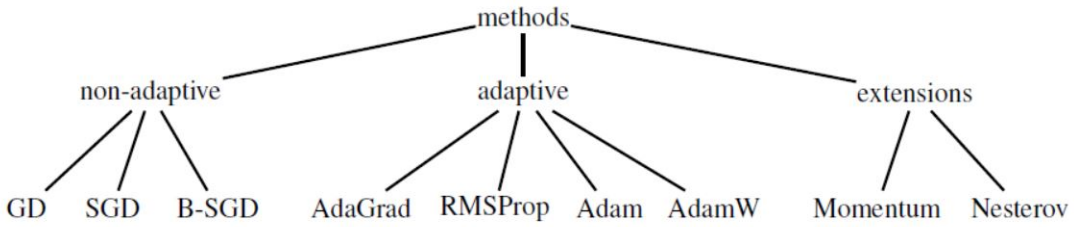
```
ADAM_W
Hyperparameters: Step_Size = 1.000000e-01, Tolerance = 1.000000e-03, Beta1 = 5.000000e-01,
                 Beta2 = 8.000000e-01, Delta = 1.000000e-07, L2_Regularization = 1.000000e-02,
                 Decay Rate = 1.000000e-02

Number of steps taken to converge with AdamW : 125
Convergence point x* = (-3.070597e-04,-1.923296e-04)
Optimized Objective at x* = 3.699069e-08
```

Figure 21: Convergence information of AdamW

CONCLUSION

The algorithms we saw above can be classified into the three sections below [Tsc19]:

- Optimizers without adaptive learning rates
- Optimizers with adaptive learning rates
- Extensions



There are many more descent algorithms in the field of Optimization, but the ones expounded in [Tsc19] are the cornerstones of the Gradient Descent methods used in fields like Machine Learning, AI etc. The observations made during the scripts' execution(in Matlab) are :

- The gradient descent methods without adaptive gradient converge slowly as compared to their counterparts, especially if there is a huge difference in the gradients calculated along different directions.
- Both of the momentum algorithms converge faster than regular gradient descent. (Note that though line-search does appear to converge faster, the step sizes are calculated with a secondary loop which contributes significantly to the time complexity.)
- AdaGrad is highly prone to overfitting, but it is very accommodating of large learning rates.
- RMSProp, which corrects the above problem is a very robust algorithm and performs well even with poor hyperparameter settings. The convergence is also very gradual, not having high jumps like the non-adaptive gradient methods.
- Adam and AdamW share all the positive qualities of RMSProp and converge significantly faster too. Both are almost equal in performance, though AdamW is slightly faster with the same hyperparameter settings.

Though RMSProp, Adam and AdamW have more hyperparameters which need more grid-search dimensions, they are highly robust to the choice of hyper-parameters. So for general problems in Convex Optimization, the above three algorithms are very well-suited. Because of the existing algorithms' exemplary results in convex optimization, current research is more focused in the area of "Non-Convex Optimization techniques".

References

[Bas+18] Amitabh Basu et al. "Convergence guarantees for rmsprop and adam in non-convex optimization and their comparison to nesterov acceleration on autoencoders". arXiv preprint arXiv:1807.06766, 2018.

[BBK08] Alexander M Bronstein, Michael M Bronstein, and Ron Kimmel. *Numerical geometry of non-rigid shapes*. Springer Science Business Media, 2008.

[BL17] Nikhil Buduma and Nicholas Locascio. *Fundamentals of deep learning: Designing next-generation machine intelligence algorithms*. O'Reilly Media Inc., 2017.

[Bot10] Léon Bottou. "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[BV14] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2014.

[DHS10] John Duchia, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research 12.Jul*. 2010, pp. 2151–2159.

[Goo+16] Ian Goodfellow et al. *Deep learning. Vol. 1*. MIT press Cambridge, 2016.

[JZ13] Rie Johnson and Tong Zhang. "Accelerating stochastic gradient descent using predictive variance reduction". In: *Advances in neural information processing systems*. 2013, pp. 315–323.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". arXiv preprint arXiv:1412.6980, 2014.

[LH19] Ilya Loshchilov and Frank Hutter. "Decoupled Weight Decay Regularization". In: *International Conference on Learning Representations*. 2019. URL: `https://openreview.net/forum?id=Bkg6RiCqY7`.

[Mez10] Juan C Meza. "Steepest descent". In: *Wiley Interdisciplinary Reviews: Computational Statistics 2.6*. 2010, pp. 719–722.

[MPS13] Athanasios Migdalas, Panos M Pardalos, and Sverre Storøy. *Parallel computing in optimization. Vol. 7*. Springer Science Business Media, 2013.

[Nes83] Yurii E Nesterov. "A method for solving the convex programming problem with convergence rate O (1/kˆ 2)". In: *Dokl. Akad. Nauk SSSR. Vol. 269*. 1983, pp. 543–547.

[Ng04] Andrew Y Ng. "Feature selection, L_1 vs. L_2 regularization, and rotational invariance". In: *Proceedings of the twenty-first international conference on Machine learning*. 2004, p. 78.

[Pol64] Boris T Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics 4.5*. 1964, pp. 1–17.

[Sut+12] Ilya Sutskever et al. "Lecture 6.5-RMSProp, COURSERA: Neural networks for machine learning". In: *University of Toronto, Technical Report*. 2012.

[Sut+13] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *International conference on machine learning*. 2013, pp. 1139–1147.

[Tsc19] Matthias Tschöpe. "Beyond SGD: Recent improvements of Gradient Descent Methods". Technische Universität Kaiserslautern in cooperation with DFKI Kaiserslautern, 2019.

[WWB18] Rachel Ward, Xiaoxia Wu, and Leon Bottou. "AdaGrad stepsizes: Sharp convergence over nonconvex landscapes, from any initialization". arXiv preprint arXiv:1806.01811, 2018.