Evolution of Optimization Algorithms

Minh Hoang, Edward Qin, Ege Caglar
University of Washington
{minh257, edwardcq, cagege}@cs.washington.edu

Project Summary

Project Scope

For this project, our main goal is to study and summarize a variety of first-order optimization algorithms derived from SGD, most notably those that have been tackling the issue of a constant learning rate not being adaptable to complex loss landscapes. Since there has been an explosion of new algorithms in this field, we pick certain directions that we believe are interesting and representative of how the field evolved over time. For each cluster of methods, this project explains how the relevant ideas first emerged and how they were built upon over time. The main theorems involved are based on generalization bounds and convergence rates of different methods.

Methodology

We selected literature by reading through the newest and most cited optimization papers on PapersWithCode and recursively reviewing related works referenced by the paper. We read through papers stemming from SGD to Adaptive Gradient methods, and from Adam variations to other, modern papers that take a different trajectory in optimization.

For each paper, we focus on how each optimization method works, especially on their update rules and how they improve upon the work of their predecessors. We then classify the optimization methods as outlined in the scope to better understand the motivation behind the direction of optimization methods over time.

Results

Our project was successful. We provided a summary of 30 optimization methods, grouped by similar time period and methodology. We also produce an evolutionary tree diagram demonstrating this.

The study helped us gain an insight into how optimization methods developed from non-adaptive methods to adaptive methods, and more recently, other methods that redefine the optimization problem or are tailored towards specific network architectures. The result of our project can be a helpful cheat sheet for engineers to choose optimization algorithms that best fit their projects' setup, or possibly a good starting point for the development of a new optimization algorithm. It is still unknown what optimization methods could be developed in the future, and we hope our project can be further extended as more optimizers are introduced.

What was Easy

Some of the things that were on the easy side of our project were the search of many recent optimization papers, based on the vast quantity of them, and how later methods refer to others in their experiment setup or related works section. For each paper, it was also quite straightforward to classify into separate sections based on the methodology proposed.

What was Difficult

Some of the most intense tasks include ensuring that the naming conventions are consistent among closely related methods. Moreover, since the number of papers we summarized is huge and many of them require advanced knowledge of optimization theory, we had to read selectively to get the main contributions and algorithm differences.

1 Introduction

For many years, gradient descent methods have been widely used to optimize Deep Learning models. One of the most well-known optimization algorithms is Stochastic Gradient Descent (SGD) [1]. However, SGD has many drawbacks when dealing with deeper and more complex models. This has led to the invention of increasingly advanced optimization algorithms, with one of the latest being Lion on February 2023 [2]. Additionally, Deep Learning has widely been applied to different fields, such as Computer Vision, Natural Language Processing, and Reinforcement Learning. For each field, there are also different field-specific optimization algorithms for each type of model. In this paper, we summarize the history of first-order optimization methods, starting from SGD and branching out into various adaptive gradients, and ending at the very latest algorithm. For each method, we discuss its update rules and how it addresses specific challenges their predecessors faced in training deep networks.

2 Project Scope

Almost all of the first-order methods used in deep learning improve on gradient descent in some form, which is why we would like to see how each method overcomes an issue with using gradient descent. We want to see how optimization algorithms evolve over time and why people tend to use one over another for some specific tasks. Our goal is to explain and summarize these advancements in a single document, as a general reference. Since there has been an explosion of new algorithms in this field, we pick certain directions that we believe are interesting and representative of how the field evolved over time. For each group, we first explain how the relevant ideas first emerged and how they evolved over time.

There have been numerous optimization algorithms proposed in the last decade, so we split them into different groups based on their proposed time and which methods they are based on:

1. Non-Adaptive methods: SGD [1], Momentum [3], NAG [4, 5].
2. Adaptive methods: AdaGrad [6], AdaDelta [7], RMSProp [8], Adam and AdaMax [9].
3. Variations of Adam: NAdam [10], AdamW [11], M-SVAG [12], AdaFactor [13], AMSGrad [14], Padam [15], Yogi [16], AdaBound [17], RAdam [18], diffGrad [19] TAdam [20], AdaBelief [21], HyAdamC [22], Lion [2].
4. Other methods: LARS [23] and its variations (LAMB [24], Fromage [25]), PowerSign and AddSign [26], LookAhead [27], SAM [28] and its variations (ASAM [29], ESAM [30]), PUGD [31].

3 Methodology

We selected literature by reading through the newest and most cited optimization papers on PapersWithCode and recursively reviewing related works referenced by the paper. Our first motivation came from the paper of Lion [2], as it is current the optimizer with the best Top-1 accuracy on ImageNet image classification. From there, we first focused on looking for papers of adaptive methods only. However, as we read more papers, we noticed that many adaptive optimizers are also compared with multiple other methods that are not variations of Adam, such as SAM [28] or LARS [23]. Therefore, we expanded our search by also researching the optimizers currently implemented in the PyTorch and TensorFlow APIs to compare optimizers that are currently used in practice. We also asked ChatGPT for extra opitmization algorithms that we could have missed in a specific time period. While we cannot possibly cover all the latest optimization methods, we selected the key papers we thought that led historical changes in research directions, as well as the latest, most interesting papers.

For each paper, since we are conducting a general survey on the history of optimization methods, we focus on how each optimization method works, especially on their update rules and how they improve upon the work of their predecessors. We then classify the optimization methods as outlined in the scope to better understand the motivation behind the direction of optimization methods over time.

4 Non-Adaptive Optimization Algorithms

In the mid-to-late 20th century, optimization methods delved into SGD and momentum. Starting from SGD, these methods generally defined a fixed step size or a scheduled step size, but did not adapt the step size over iterations. However, the methods were groundbreaking and formed the foundation for many methods today.

4.1 Stochastic Gradient Descent (SGD)

In 1951, Robbins and Munro introduced **SGD** [1], a groundbreaking paper that many optimizers today are based on. In their paper, Robbins and Munro establish that stochastic approximation can converge, and can also predict optimal values with less assumptions on the nature of the function (i.e. linear regression in a certain number of dimensions).

They define a simple update rule:

$$\theta_t = \theta_{t-1} - a_{t-1}(\alpha - y_{t-1})$$

where θ_i is the sequence of guesses for the input of the function, α is the minimum value of $f(\theta)$ at the root θ, y_i are like labels of the inputs, and a_i are pre-defined step sizes.

The algorithm is formalized and generalized with a global step size as shown below:

Algorithm 1: Stochastic Gradient Descent

Data: stochastic function $f(\theta)$, initial parameter vector θ_0, step size α
Result: θ_t

$t \leftarrow 0$
while θ_t *is not converged* **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$ `/* Compute gradient */`
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot g_t$ `/* Update weight */`
end
return θ_t

4.2 Momentum

In 1964, Polyak introduced the classical **Momentum** method [3]. The motivation behind momentum was to speed up SGD, where momentum remembers the weight update at each iteration using an exponential decay factor. When we average over past gradients, the velocity at which the optimization steps increases when the past gradients are similar, whereas the velocity decreases when the past gradients oscillate.

The momentum method employs the following update rules to the velocity and weight vectors:

$$v_t = \beta \cdot v_{t-1} - \alpha \nabla_\theta f(\theta_t)$$

$$\theta_t = \theta_{t-1} + v_t$$

where β is the decay rate, and α is the step size.

4.3 Nesterov's Accelerated Gradient

In 1983, Nesterov introduced his **Accelerated Gradient Descent** (NAG) [4]. While similar to classical momentum, Nesterov's method employs a very slight change to the velocity vector update rule:

$$v_t = \beta \cdot v_{t-1} - \alpha \nabla_\theta f(\theta_{t-1} + \beta \cdot v_{t-1})$$

In classical momentum, the gradient update is first computed, then added to θ_t. However, Nesterov's method intuitively first takes a step along the velocity vector, then corrects the error using the gradient on the vector after the step, $\nabla f(\theta_{t-1} + \beta \cdot v_{t-1})$. This allow the computation to become much more stable. In fact, Sutskever proved that NAG outperforms classical momentum since it can adjust and have smaller effective momentum in high-curvature eigen-directions [5]. NAG also achieves a better convergence rate of $O(1/T^2)$ over the SGD convergence rate of $O(1/T)$.

5 Adaptive Optimization Algorithms

In the early 2010s, adaptive gradient methods became increasingly popular. The concept of adapting the effective learning rate over iterations to take larger steps over areas of the same gradient or smaller steps closer to the optimum shaped many new optimization methods. The concepts were built upon and ultimately culminated into the concept of Adam.

5.1 AdaGrad

Duchi et al. were one of the first to introduce adaptive subgradient methods through **AdaGrad** in 2010 [6]. The goal was to use dynamic incorporation of characteristics of geometry of data for more informative gradient-based learning. The method works well on sparse data and parameters because the algorithm adapts individual per-parameter learning rates such that frequent features are assigned lower learning rates and infrequent features are assigned high learning rates.

Generalized, the update step is:

$$\theta_t = \Pi_{\mathcal{X}}^{G_{t-1}^{1/2}}(\theta_{t-1} - \alpha \cdot G_{t-1}^{-1/2} g_{t-1})$$

where G_t is defined as the outer product matrix of the subgradients g_τ up to timestep t, $G_t = \sum_{\tau=1} t g_\tau g_\tau^T$, and $\Pi_{\mathcal{X}}^{G_t^{1/2}}(y)$ is defined using the Mahalanobis norm, or the projection of point y onto $\mathcal{X}$ according to ${G_t}^{1/2}$:

$$\Pi_{\mathcal{X}}^{G_t^{1/2}}(y) = \arg\min_{x \in \mathcal{X}} \|x - y\|_{{G_t}^{1/2}} = \arg\min_{x \in \mathcal{X}} \langle x - y, {G_t}^{1/2}(x - y) \rangle$$

To simplify notation, Kingma and Ba [9] denoted the basic version of AdaGrad as:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{g_{t-1}}{\sqrt{\sum_{i=1}^{t-1} g_i^2}}$$

5.2 AdaDelta

In 2012, Zeiler introduced **AdaDelta** [7], a dynamic, robust, and insensitive to hyperparameters gradient descent method. Derived from AdaGrad, AdaDelta aimed to correct for the continual decay of learning rates (becoming infinitesimally small) and the need for a manually selected global learning rate.

To prevent continual decay, AdaDelta changed AdaGrad's accumulation of squared gradients over time to that of a window in time. However, to efficiently store the previous squared gradients, it instead stored an exponentially decaying running average of the squared gradients with decay rate β, defined as:

$$E[g^2]_t = \beta \cdot E[g^2]_{t-1} + (1 - \beta) g_t^2$$

Then dividing by the square root of the exponentially decaying average, Zeiler essentially calculates the RMS of previous squared gradients where:

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

with some regularization constant ϵ. Then in rearranging Newton's method and assuming a diagonal Hessian, Zeiler removed the need for a global learning rate and defines the weight update at each time step as:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} \cdot g_t$$

In this update rule, the numerator represents the acceleration term over previous gradients while the denominator represents the squared gradient information from AdaGrad.

5.3 RMSProp

In 2012, Tieleman and Hinton introduced the **RMSProp** algorithm [8] that divided the gradient by a running average of its recent magnitude. RMSProp was inspired by RProp, which only updates learning rate based on the sign of the gradient (multiplicatively increase or decrease if the signs of the last two gradients agree or disagree). RMSProp aimed to combine the robustness of RProp, the efficiency of mini-batching, and the effective averaging of gradients over mini-batches.

RMSProp defines a moving average on the squared gradient of each weight and divides the gradient by $\sqrt{MeanSquare(w, t)}$ in each update step:

$$MeanSquare(w, t) = \beta \cdot MeanSquare(w, t-1) + (1 - \beta) \cdot (\nabla_\theta f(\theta_{t-1}))^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{MeanSquare(w, t-1)}} \cdot \nabla_\theta f(\theta_{t-1})$$

5.4 Adam and AdaMax

In 2014, Kingma and Ba [9] introduced the **Adam** optimizer. The goal was to optimize stochastic objectives with high-dimensional parameter spaces, using only the first-order gradient and little memory. Adam was based on a combination of AdaGrad (used for sparse gradients, like Adam with parameter $\beta_1 = 0$) and RMSProp (used for online, non-stationary setting).

Algorithm 2: Adam

Data: stochastic function $f(\theta)$, initial parameter vector θ_0, step size α, decay rate β_1, decay rate β_2, regularization constant ϵ

Result: θ_t

1 $m_0 \leftarrow 0$ /* First momentum vector */

2 $v_0 \leftarrow 0$ /* Second momentum vector */

3 $t \leftarrow 0$

4 **while** θ_t *is not converged* **do**

5 $\quad t \leftarrow t + 1$

6 $\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ /* Compute gradient */

7 $\quad m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ /* Update first momentum */

8 $\quad v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ /* Update second momentum */

9 $\quad \hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ /* Bias-correct first momentum */

10 $\quad \hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ /* Bias-correct second momentum */

11 $\quad \theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$

12 **end**

13 **return** θ_t

Adam works by calculating the exponential weighted average of the first and second momentum vectors, as well as applying correction for initialization bias. β_1 describes the decay rate of the previous first momentum vectors, and β_2 describes the decay rate of the previous second momentum.

Another important component of Adam is the adaptive choice of step size (adaptive learning rate). In each timestep t, the effective step size taken is $\Delta_t = \alpha \cdot \hat{m}_t / \sqrt{\hat{v}_t}$. This is desirable because as the weight parameters approach the optimum, $\hat{m}_t / \sqrt{\hat{v}_t}$ (the signal-to-noise ratio) approaches 0, and the optimizer takes smaller effective steps.

AdaMax is an extension on Adam where the second momentum is replaced by a p^{th} momentum where $p \to \infty$. We then replace v_t in Algorithm 2 with:

$$u_t = \max(\beta_2 \cdot u_{t-1}, |g_t|)$$

as the update for the p^{th} momentum vector, and we no longer need to correct for initialization bias in lines 9 and 10. Additionally, the effective step size in line 11 is modified to use the biased-correction term of the first momentum:

$$\theta_t \leftarrow \theta_{t-1} - (\alpha / (1 - \beta_1^t)) \cdot m_t / u_t$$

6 Variations of Adam

After its introduction, the Adam optimizer drew worldwide attention due to its effectiveness in handling noisy and sparse datasets, which are very common in the real world. Since it can greatly outperform SGD and its variants, Adam is still widely used in training many deep learning models. However, there are still several circumstances where Adam fails to converge. Therefore, many variations of Adam have also been proposed to improve training results, as well as to overcome several drawbacks of it.

6.1 NAdam

One of the very first variations of Adam was **NAdam**, which was proposed by Dozat [10] in 2016. NAdam incorporates Nesterov's accelerated gradient to speed up Adam. In classical momentum, we make a step in the previous momentum vector and a step in the current gradient. By rewriting the update step, we can compute the next momentum step in the current step. This rewrite also accounts for initialization bias correction. We can then easily modify the algorithm to use Nesterov's accelerated gradient. In order to transform Adam into NAdam, we make a slight modification to line 9 of

Algorithm 2 to include Nesterov's momentum:

$$\hat{m_t} \leftarrow (\mu_{t+1} \cdot m_t/(1 - \prod_{i=1}^{t+1} \mu_i)) + ((1 - \mu_t) \cdot g_t/(1 - \prod_{i=1}^{t} \mu_i)) \qquad (\mu_t \text{ is defined by a momentum scheduler})$$

Additionally, in his implementation, the author set $\beta_1 = 0.99$ and, based on analysis on Nesterov momentum from Sustskever et al. [5], defined:

$$\mu_t \leftarrow \beta_1\big(1 - \frac{1}{2} \cdot 0.96^{t \cdot \psi}\big)$$
$$\mu_{t+1} \leftarrow \beta_1\big(1 - \frac{1}{2} \cdot 0.96^{(t+1) \cdot \psi}\big) \qquad (\psi \text{ is the momentum decay})$$

6.2 AdamW

Shortly after the introduction of NAdam, in November 2017, Loshchilov and Hutter proposed another variation of Adam, called **AdamW** [11]. The main contribution of this proposal was to decouple weight decay for SGD and Adam. In normal SGD, ℓ_2 regularization is the same as weight decay (where λ weight decay is equivalent to ℓ_2 regularization with hyperparameter $\lambda' = \lambda/\alpha$ with step size α). However, this is not the case when the gradient is adapted, since ℓ_2 on large gradients scales down the gradient of the regularizer.

To apply weight decoupling, we move the regularization from the gradient calculation step to the weight update step. This allows us to only adapt the gradients of the loss function, and we can regularize weights using the same λ. Specifically, in order to get from Adam to AdamW, we modify line 11 of Algorithm 2 to include decoupled weight decay as follows:

$$\theta_t \leftarrow \theta_{t-1} - \eta_t \cdot (\alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon) + \boldsymbol{\lambda \theta_{t-1}})$$

where η_t is a scaling factor defined by some user-defined procedure $SetScheduleMultiplier(t)$.

6.3 M-SVAG

In Febraury 2018, Balles and Hennig [12] investigated multiple circumstances on why sometimes Adam fails to work well by dissecting it into 2 aspects and analyzing them separately: The update direction of each weight and the update magnitude. Their proposed solution, **M-SVAG**, short for *Momentum Stochastic Variance-Adapted Gradient*, makes use of the sign of the update direction and the relative variance of the update magnitude to achieve better generalization than Adam.

In their paper, the authors divided Adam into a combination of 2 aspects: The update direction, given by the sign of m_t, and the update magnitude, which is approximated by the stepsize α and the relative variance η_t between v_t and m_t. The goal of using the sign as the update direction for the gradient is to deal with noisy, ill-conditioned problems with diagonally dominant Hessians. Thus, applying variance adaptation to the sign update guarantees convergence without manually decreasing the global step size. To get M-SVAG, we add the following computations to Algorithm 2 before updating θ_t:

$$s_t \leftarrow (1 - \rho(\beta_2, t))^{-1}(\hat{v}_t - \hat{m}_t)^2 \qquad (\rho(\beta_2, t) = \tfrac{(1-\beta_2)(1+\beta_2^{t+1})}{(1+\beta_2)(1-\beta_2^{t+1})})$$
$$\gamma_t \leftarrow \hat{m}_t^2/(\hat{m}_t^2 + \rho(\beta_2, t) \cdot s_t)$$

Then, our update rule will be:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot (\gamma_t \cdot \hat{m}_t)$$

6.4 AdaFactor

In April 2018, Shazeer and Stern proposed **AdaFactor** [13]. The goal of AdaFactor is to reduce memory usage while still retaining empirical adaptivity of Adam. In Adam, maintaining second-moment estimators requires an amount of memory equal to the number of parameters, which is fairly costly. To resolve this, the authors suggest to track moving averages of the weight matrix in order to reconstruct a low-rank approximation of the exponentially smoothed accumulator at each training step that is optimal with respect to the generalized Kullback-Leibler divergence. This helps reduce the memory requirement for a $n \times m$ matrix from $\mathcal{O}(nm)$ to $\mathcal{O}(n + m)$.

AdaFactor is defined in terms of relative step sizes $\{\rho_t\}_{t=1}^T$, which is then multiplied by the RMS of the parameter vector's components. It also scales down the updates of the weight vectors whenever the RMS exceeds a clipping

threshold d. Additionally, β_1 is set to 0 in AdaFactor, as the memory requirements of the second-moment accumulator has been reduced.

Algorithm 3: AdaFactor for weight vectors

Data: stochastic function $f(\theta)$, initial parameter vector θ_0, relative step sizes ρ_t, decay rate β_2, clipping threshold d, regularization constants ϵ_1, ϵ_2

Result: θ_t

1 $t \leftarrow 0$

2 **while** θ_t *is not converged* **do**

3 $\quad t \leftarrow t + 1$

4 $\quad \alpha_t \leftarrow \max(\epsilon_2, \text{RMS}(\theta_{t-1}))\rho_t$ `/* Scale relative step size */`

5 $\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ `/* Compute gradient */`

6 $\quad \hat{v}_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (g_t^2 + \epsilon_1 \cdot \mathbb{1}_n)$ `/* Update reduced second moment */`

7 $\quad m_t \leftarrow g_t / \sqrt{\hat{v}_t}$

8 $\quad \hat{m}_t \leftarrow m_t / \max(1, \text{RMS}(m_t)/d)$ `/* Update clipping */`

9 $\quad \theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot \hat{m}_t$

10 **end**

11 **return** θ_t

For weight matrices settings, we also keep track of the moving averages of row and column sums of the squared gradients for matrix-valued variables, r_t and c_t, as follows:

$$r_t \leftarrow \beta_2 r_{t-1} + (1 - \beta_2) \cdot (g_t^2 + \epsilon_1 \cdot \mathbb{1}_n \cdot \mathbb{1}_m^T) \cdot \mathbb{1}_m$$
$$c_t \leftarrow \beta_2 c_{t-1} + (1 - \beta_2) \cdot \mathbb{1}_n^T \cdot (g_t^2 + \epsilon_1 \cdot \mathbb{1}_n \cdot \mathbb{1}_m^T)$$

Then, our reduced second moment update rule will be:

$$\hat{v}_t \leftarrow r_t \cdot c_t / \mathbb{1}_n^T \cdot R_t$$

AdaFactor is able to achieve comparable performances with Adam with very little auxiliary storage in the optimizer.

6.5 AMSGrad

One of the problems that Adam faces is that it may fail to converge even in a simple convex setting. One variation that was proposed to fix the convergence issue of Adam is **AMSGrad** by Reddi et al. [14]. In this paper, the authors provide a specific convex optimization example where both RMSProp and Adam fail to converge to an optimal solution. To deal with this, AMSGrad updates the parameters θ_t using the maximum of past squared gradients v_t instead of the exponential average. Specifically, we make the following change to line 9 of Algorithm 2 to get AMSGrad:

$$\hat{v}_t \leftarrow max(\hat{v}_{t-1}, v_t)$$

6.6 Padam

Adam and AMSGrad are widely used, and shortly after the proposal of AMSGrad, an extra argument to activate AMSGrad has been included in the PyTorch implementation of Adam. However, in June 2018, Chen and Gu [15] argued that these 2 optimizers can sometimes be over-adaptive, meaning that as their initial learning rates are relatively smaller than SGD, Adam and AMSGrad have worse generalization compared to SGD on larger models. To fix this, the authors proposed **Padam**, short for *Partially adaptive momentum estimation method*, which inherits the fast convergence rate of Adam while achieving better generalization. The key difference between Padam and AMSGrad is an inclusion of a partially adaptive parameter $p \in (0, 1/2]$. While m_t is still the first order momentum as in AMSGrad, it is now partially adapted in our update rule. When $p \to 0$, Padam gradually converts to SGD with momentum and when $p = 1/2$, Padam is exactly AMSGrad. From AMSGrad, we get Padam by making an additional change to line 11 of Algorithm 2:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{\boldsymbol{v}}_{\boldsymbol{t}}^{\boldsymbol{p}}} + \epsilon)$$

6.7 Yogi

One of the limitations that AMSGrad and Padam face is that they only work for convex functions. For non-convex optimization problems, these optimizers fail to converge. This happens when the second momentum update v_t blows up. In order to fix this issue, Zaheer et al. [16] proposed a new way to update and initialize v_t that works even for

non-convex settings. In the update step, v_t tends to lose control of past values quickly whenever g_t^2 has high variances or when updates are too sparse. To fix this, the author introduced **Yogi** in December 2018, which makes an update on the second momentum by taking the sign of the update step and multiplying it with g_t^2. This helps Yogi removes the dependency on v_{t-1} of Adam and the previous variations and only depends on g_t^2. To transform Adam to Yogi, we change line 8 of Algorithm 2 as follows:

$$v_t \leftarrow v_{t-1} - (1-\beta_2) \cdot \mathbf{sign}(\boldsymbol{v_{t-1} - g_t^2}) \cdot \boldsymbol{g_t^2}$$

6.8 AdaBound and AMSBound

Another case where Adam and even AMSGrad may fail to converge is when learning rates are too extreme or unstable (below 0.01 or over 1000, for example). Therefore, in February 2019, Luo et al. proposed an improvement of these two methods, **AdaBound** and **AMSBound** [17], which employed dynamic bounds on the learning rates. Given the lower and upper bound learning rate functions $\eta_l(t)$ and $\eta_u(t)$, initialized to 0 and ∞ as $t = 0$ respectively, AdaBound and AMSBound apply the concept of gradient clipping on learning rates $\hat{\eta}_t$ element-wisely so that the output is constrained within the boundary functions. We add the following to Algorithm 2 before making our update step:

$$\hat{\eta}_t \leftarrow \text{Clip}(\alpha/\sqrt{\hat{v}_t}, \eta_l(t), \eta_u(t))$$
$$\eta_t \leftarrow \hat{\eta}_t/\sqrt{t}$$

Then, our update step of AdaBound and AMSBound will be:

$$\theta_t \leftarrow \theta_{t-1} - \eta_t \cdot \hat{m}_t$$

6.9 RAdam

In late 2019, Liu et al. [18] discovered another reason for Adam's inability to converge or converge to bad optima, especially in training Transformer-based models. Because of the lack of training samples in the early stages, the adaptive learning rate of Adam has large variance. One possible solution to this problem is using learning rate warmup as a variance reduction technique: Starting the training loop with a very small learning rate for the first few epochs and then gradually increasing until the desired learning rate as the number of epoch increases.

With that, the authors proposed **RAdam**, short for *Rectified Adam*, which adapted learning rate warmup to reduce variance in the early stage. Instead of calculating the bias-corrected moving average for second momentum, RAdam first approximates the exponential moving average (EMA) using a simple moving average (SMA):

$$p\left(\frac{(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\cdot g_i^2}{1-\beta_2^t}\right) \approx p\left(\frac{\sum_{i=1}^{f(t,\beta_2)} g_{t+1-i}^2}{f(t,\beta_2)}\right) \qquad (f(t,\beta_2) \text{ is the length of the SMA})$$

After that, it calculates the maximum and length of the approximated SMA. Then, it computes the adaptive learning rate and uses first-order approximation to calculate the variance rectification term before making the update step. Particularly, in Algorithm 2, we replace line 10 and 11 with the following pseudocode:

Algorithm 4: RAdam - update rule using adaptive learning rate

$\rho_\infty \leftarrow 2/(1-\beta_2) - 1$ `/* Max. length of approximated SMA */`
$\rho_t \leftarrow \rho_\infty - 2t\beta_2^t/(1-\beta_2^t)$ `/* Length of approximated SMA */`
if $\rho_t > 4$ **then**
  $l_t \leftarrow \sqrt{(1-\beta_2^t)/v_t}$ `/* Adaptive learning rate */`
  $r_t \leftarrow \sqrt{\frac{(\rho_t-4)(\rho_t-2)\rho_\infty}{\rho_\infty-4)(\rho_\infty-2)\rho_t}}$ `/* Variance rectification term */`
  $\theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot r_t \cdot \hat{m}_t \cdot l_t$
else
  $\theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot \hat{m}_t$
end

By setting up a condition on whether or not the variance is tractable ($\rho_t > 4$), RAdam can deactive adaptive learning rate when its variance is divergent, thus avoiding undesired instability in the first few updates.

6.10 diffGrad

In December 2019, Dubey et al. introduced **diffGrad** [19], a novel method that took advantage of local changes in gradients. In diffGrad, the step size of each parameter is adjusted accordingly so that it matches the speed of the

parameter's gradient changing. When the gradient change falls within a certain range, it clamps down on the step size in order to ensure that if it's a global minima it won't overshoot like in Adam, AMSGrad or any previous adaptive learning optimizers. In order to do so, it introduces an new friction coefficient ξ_t to control the learning rate using information of short-term gradient behavior. This friction coefficient helps reduce redundant learning and increase the rate of convergence. It also helps in finding an optimum solution by reducing the vertical fluctuation near local optima. In Algorithm 2, along with calculating m_t and v_t, diffGrad will also calculate ξ_t as follow:

$$\xi_t \leftarrow 1/(1 + e^{-|g_{t-1} - g_t|})$$

Then, the update rule will become:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \boldsymbol{\xi_t} \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$$

With diffGrad, the step size can be more flexible, which allows better training results, especially in CNNs.

6.11 TAdam

In early 2020, Ilboudo et al. [20] proposed another variation of Adam called **TAdam** that is more robust against noise and diverse attacks, especially in robotics domain. The key point of TAdam is its ability to detect outliers and discard them when needed using the robust student-t distribution, a well-known robust probability distribution instead of EMA.

To replace EMA with the student-t distribution, TAdam includes a degree of freedom variable ν to control robustness, which is equal to the dimension of the gradient. Then it changes the update rule of the first momentum by replacing line 7 of Algorithm 2 with the following:

Algorithm 5: TAdam - update first momentum rule

1 $W_0 \leftarrow \beta_1/(1 - \beta_1)$
2 **while** θ_t *is not converged* **do**
3 $\quad w_t \leftarrow (\nu + d)\left(\nu + \sum_j \frac{(g_t^j - m_{t-1}^j)^2}{v_{t-1}+\epsilon}\right)^{-1}$
4 $\quad m_t \leftarrow \frac{W_{t-1}}{W_{t-1}+w_t} \cdot m_{t-1} + \frac{w_{t-1}}{W_{t-1}+w_t} \cdot g_t$
5 $\quad W_t \leftarrow (2\beta_1 - 1)/\beta_1 \cdot W_{t-1} + w_t$
6 **end**

6.12 AdaBelief

So far, while all variations of Adam have achieved somewhat better accuracy than Adam, they still underperform on large-scale datasets such as ImageNet, compared to SGD. Furthermore, when training generative models such as GANs, these optimizers tend to become empirically unstable. For that, Zhuang et al. proposed **AdaBelief** [21], a variation that solves all the aforementioned problems: maintaining training stability for generative models (GANs) and good generalization on large-scale datasets like SGD, while still having fast convergence rate like Adam.

The intuition behind AdaBelief's implementation is the choice of the stepsize based on the current gradient direction. By considering the EMA of the gradient as the prediction of the gradient at the next time step, depending on the distance between the current gradient and the prediction, the step size AdaBelief makes will vary. If the distance is large, the step size will be small and vice versa. Instead of measuring the EMA of g_t^2 like in Adam, AdaBelief measures the EMA of $(g_t - m_t)^2$ in order to take different step sizes accordingly. To get from Adam to AdaBelief, we make one change to line 8 of Algorithm 2:

$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot \boldsymbol{(g_t - m_t)^2} + \boldsymbol{\epsilon}$$

6.13 HyAdamC

In June 2021, Kim et al. [22] proposed a hybrid first-order optimization algorithm that is based on Adam to improve training CNN efficiency. **HyAdamC** uses three new velocity control functions to adjust its search strength carefully in term of initial, short, and long-term velocities. Moreover, it utilizes an adaptive coefficient computation method to prevent that a search direction determined by the first momentum is distorted by any outlier gradients.

In order for the method to be more robust, HyAdamC first modifies the computation of the first momentum by checking the difference between m_t and g_t and increase β_1 in proportion to the difference. Then, in order to find the optimal θ_t, HyAdamC collects various information about the current optimization terrain from the past and current gradients, and utilizes them using 3 adaptive control functions:

- Initial-Term Velocity Control Function: Controls the degree of convergence in the first few epochs, which is very similar to the learning rate warmup strategy of RAdam. This equation is defined as follow:

$$\xi_I(\beta_2, \rho_t, \rho_\infty) \leftarrow \left(\frac{\rho_\infty (1-\beta_2^t)(\rho_t^2 - 6\rho_t + 8)}{\rho_t(\rho_\infty^2 - 6\rho_\infty + 8)} \right)^{\frac{1-\sum_{i=1}^{4} \delta_{\rho_t, i}}{2}} \qquad (\rho_t, \rho_\infty \text{ are defined similar to RAdam})$$

- Short-Term Velocity Control Function: Adjusts the search velocity based on the change in gradient between g_t and g_{t-1}. It is formulated as:

$$\xi_S(g_t, g_{t-1}) \leftarrow (1 + e^{-\sigma_t(|g_t - g_{t-1}| - \mu_t)})^{-1} \qquad (\sigma, \mu \text{ are mean and s.d of } |g_t - g_{t-1}|)$$

- Long-Term Velocity Control Function: Adjusts the search velocity based on all previous gradients $g_1 \dots g_{t-1}$. This control function is used as a new second momentum computation method. Specifically, we replace Line 8 of Algorithm 2 with:

$$v_t = \xi_L(v_{t-1}, m_{t-1}, g_t, \beta_1, \beta_2) \leftarrow \beta_2 \cdot v_{t-1} + \beta_{1,t-1}^2 (1-\beta_2) \cdot (m_{t-1} - g_t)^2$$

With that, before updating θ_t, HyAdamC scales the bias-corrected first momentum $\hat{m}_t$ using the remaining velocity control functions:

$$\hat{m}_t \leftarrow \xi_I(\beta_2, \rho_t) \cdot \xi_S(g_t, g_{t-1}) \cdot \hat{m}_t$$

6.14 Lion

The most recent optimizer that was proposed is **Lion** [2] by Chen et al. in February 2023. Lion is an optimizer derived from AdamW that was discovered through symbolic program search. In their paper, the authors proposed a method to formulate algorithm discovery as program search and apply it to discover optimization algorithms. First, they need to find high-quality algorithms in the infinite and sparse program space, and then further select those that generalize from small proxy tasks to much larger, state-of-the-art tasks.

In order to reach such result, several efficient search techniques have to be deployed, including but not limited to: warm-start and restart, abstract execution, funnel selection, and program simplification. Funnel selection and program simplification of AdamW led to the raw Lion. After dropping unnecessary functions that does not affect the results such as $cosh, clip, arcsin$, they arrived at Lion, a simple yet efficient optimization algorithm that makes the update rule using only the sign function. Additionally, Lion is much more memory-efficient than other optimizers, as it only needs momentum. An additional feature of Lion is that it updates θ_i first before updating $\hat{m}_t$.

Algorithm 6: Lion

Data: stochastic function $f(\theta)$, initial parameter vector θ_0, step size α, decay rate β_1, decay rate β_2
Result: θ_t

1 $m_0 \leftarrow 0$
2 **while** θ_t *is not converged* **do**
3 $\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
4 $\quad c_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$
5 $\quad \theta_t \leftarrow \theta_{t-1} - \alpha \cdot (sign(c_t) + \theta_{t-1})$
6 $\quad \hat{m}_t \leftarrow \beta_2 \cdot m_{t-1} + (1-\beta_2) \cdot g_t$
7 **end**
8 **return** θ_t

So far, a fine-tuned vision model optimized by Lion achieved the highest top-1 accuracy in image classification for ImageNet. However, despite outperforming AdamW and Adam on multiple tasks, Lion still has some limitations. For some tasks like in-context learning becnhmarks, or training with a batch size less than 64, Lion does not make an improvement compared to AdamW, while being potentially expensive for larger models due to a great amount of momentum tracking.

7 Other Optimization Algorithms

In addition to the emergence of optimization algorithms that are based on SGD or Adam, there have also been various algorithms that go in a completely different direction from those two. There has been an emergence of methods applied to specific fields like Computer Vision, Reinforcement Learning, in-context learning, as well as methods that completely redefined the optimization problem.

7.1 PowerSign and AddSign

In 2017, Bello et al. [26] applied Reinforcement Learning to automate the process of discovering optimization methods. Specifically, they trained a Recurrent Neural Network (RNN) controller that generated a series of update rules and discovered 2 families of update rules that can achieve much better performances than other original optimizers like SGD, RMSProp or Adam.

When sampling the newly generated update rules using the RNN controller, a common term that was found is the product term $\text{sign}(g) \cdot \text{sign}(m)$, which provides a binary signal on whether the direction of the gradient and its moving average agree on a single dimension. This leads to the discovery of 2 families of update rules, **PowerSign** and **AddSign**, both of which make use of the product term in different ways.

Given $f(t)$ that is either 1 or an internal decay function of step t (linear decay, cyclical or restart decays, or a combination of several decay functions):

- PowerSign scales the update of each parameter by $\alpha^{f(t)}$ or $1/\alpha^{f(t)}$ depending on the agreement between the direction of the gradient and its moving average. The general update rule of PowerSign is:
$$\theta_t \leftarrow \alpha^{f(t)\cdot\text{sign}(g_t)\cdot\text{sign}(m_t)} \cdot g_t$$
- AddSign scales the update of each parameter by $\alpha + f(t)$ or $\alpha - f(t)$ depending on the agreement between the direction of the gradient and its moving average. The general update rule of AddSign is:
$$\theta_t \leftarrow (\alpha + f(t) \cdot \text{sign}(g_t) \cdot \text{sign}(m_t)) \cdot g_t$$

Both PowerSign and AddSign are robust to slight changes in hyperparameters, including the decay rates for the moving averages, and larger α values generally lead to faster convergence rates with minor loss.

7.2 LARS and variations - LAMB and Fromage

Also in 2017, You et al. proposed **LARS** (*Layer-wise Adaptive Rate Scaling*) [23] to address how larger batch size often results in lower model accuracy in the context of large CNNs. In particular, LARS was developed for training using AlexNet-BN, where local response normalization layers were replaced with batch normalization layers.

With larger batch size, training takes less iterations, and we opt for larger step sizes, which are more susceptible to diverging. As such, LARS proposes the use of separate learning rate for each layer as opposed to each weight, and controls the magnitude of each update with respect to the weight norm.

LARS defines a local learning rate λ^ℓ for each layer ℓ, defined using a trust coefficient η determining how much we trust layers to change its weights in one update:

$$\lambda^\ell = \eta \cdot \frac{\left\|\theta^\ell\right\|}{\|\nabla_\theta L(\theta^\ell)\|}$$

We can then extend the denominator to include the weight decay β to produce the following algorithm:

Algorithm 7: LARS

Data: base step size α_0, momentum m, weight decay β, LARS coefficient η, iterations T
Result: θ_t

$v \leftarrow 0$
for each *layer* ℓ **do**
  Initialize θ_0^ℓ `/* Initialize weight for each layer */`
end
for $t = 1, 2, ..., T$ **do**
  for each *layer* ℓ **do**
    $g_{t-1}^\ell \leftarrow \nabla_\theta L(\theta_{t-1}^\ell)$ `/* Obtain stochastic gradient for mini-batch */`
    $\alpha_{t-1} \leftarrow \alpha_0 \cdot (1 - \frac{t-1}{T})^2$ `/* Global learning rate */`
    $\lambda^\ell \leftarrow \frac{\left\|\theta_{t-1}^\ell\right\|}{\left\|g_{t-1}^\ell\right\| + \beta\left\|\theta_{t-1}^\ell\right\|}$ `/* Local learning rate` λ^ℓ `*/`
    $v_t^\ell \leftarrow m v_{t-1}^\ell + \alpha_t \cdot \lambda^\ell \cdot (g_{t-1}^\ell + \beta\theta_{t-1}^\ell)$ `/* Momentum update */`
    $\theta_t^\ell \leftarrow \theta_{t-1}^\ell - v_t^\ell$
  end
end
return θ_t

Extending from LARS, You et al. developed **LAMB** (*Layer-wise Adaptive Moments*) [24] to improve optimization performance on attention models like Google's BERT . LAMB applies per-dimension normalization with respect to the square root of the second moment, as well as layerwise normalization for layerwise adaptivity. LARS and LAMB's convergence rates depend on L_{avg} instead of L_∞, and are thus better than that of SGD.

Algorithm 8: LAMB

Data: initial weight vector θ_0, learning rate $\{\alpha_t\}_{t=1}^T$, hyperparameters β_1 and β_2, scaling function ϕ, $\epsilon > 0$
Result: θ_t

$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
for $t = 1, 2, ..., T$ **do**
  for each *layer* ℓ **do**
    $g_{t-1}^\ell \leftarrow \nabla_\theta L(\theta_{t-1}^\ell)$ `/* Obtain stochastic gradient for mini-batch */`
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
    $m_t \leftarrow m_t / (1-\beta_1^t)$
    $v_t \leftarrow v_t / (1-\beta_2^t)$
    $r_t \leftarrow m_t / (\sqrt{v_t} + \epsilon)$ `/* Compute ratio */`
    $\theta_t^\ell \leftarrow \theta_{t-1}^\ell - \alpha_t \frac{\phi(\|\theta_{t-1}^\ell\|)}{\|r_{t-1}^\ell + \lambda\theta_{t-1}^\ell\|}(r_{t-1}^\ell + \lambda\theta_{t-1}^\ell)$
  end
end
return θ_t

Then, in 2021, Bernstein et al. introduced another variation of LARS, **Fromage** (*Frobenius matched gradient descent*) [25], along with the concept of distance between neural networks called deep relative trust. While similar to LARS, Fromage works well across multiple standard neural networks including GANs and natural language transformers, without the need of learning rate tuning.

Deep relative trust penalizes the relative size of perturbations to each layer, so Fromage bounds layerwise perturbations such that $\|\Delta W_\ell\|_F / \|W_\ell\|_F \leq \alpha$. Additionally, compounding growth on weight norms in layers that involve batch norm is corrected using an additional weight decay hyperparameter in LARS, whereas Fromage uses an explicit correction using the Frobenius norm.

Algorithm 9: Fromage

Data: learning rate α, matrices $\{W_\ell\}_{\ell=1}^L$
Result: $\{W_\ell\}_{\ell=1}^L$

$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
while θ_t *is not converged* **do**
  $\{g_t\}_{\ell=1}^L \leftarrow$ gradients on matrices
  for *layer* $\ell = 1$ **to** L **do**
    $W_\ell \leftarrow \frac{1}{\sqrt{1+\alpha^2}}(W_\ell - \alpha \cdot \frac{\|W_\ell\|_F}{\|g_t\|_F} \cdot g_\ell)$
  end
end
return $\{W_\ell\}_{\ell=1}^L$

7.3 LookAhead

Normally, to improve SGD, there are 2 main approaches: Using adaptive learning rates schemes (AdaGrad, Adam,...) and using accelerated schemes (Momentum, NAG). However, in June 2019, Zhang et al. [27] proposed **LookAhead**, an optimizer that goes in a completely different direction from SGD or Adam. Specifically, the author described LookAhead's direction as *orthogonal* to SGD and Adam.

Unlike SGD and Adam, LookAhead keeps track of 2 sets of weight: a set of *fast weights* ϕ, generated by another optimizer, and a set of *slow weights* θ. It makes the choice of direction by "looking ahead" of the set of fast weights generated by a different optimizer before updating the slow weights once the direction has been chosen. Although this

requires an extra inner for loop, LookAhead has been proved to significantly improve the performances of SGD and Adam when ran on top of them, with lower variance, negligible computational and memory cost, as well as being robust to hyperparameter tuning.

Algorithm 10: LookAhead

Data: stochastic function $f(\theta)$, initial parameter vector θ_0, slow weight step size α, optimizer A, synchronization period k
Result: θ_t

for $t = 1, 2, ...$ **do**
  $\phi_{t,0} \leftarrow \theta_{t-1}$ `/* Synchronize fast weights */`
  for $i = 1, 2, ..., k$ **do**
    $\phi_{t,i} \leftarrow \phi_{t,i-1} + A(f, \phi_{t,i-1})$ `/* Inner update fast weights */`
  end
  $\theta_t \leftarrow \theta_{t-1} + \alpha \cdot (\phi_{t,k} - \theta_{t-1})$ `/* Outer update slow weights */`
end
return θ_t

7.4 SAM and variations - ASAM and ESAM

In 2020, Foret et al. reformulate gradient descent using the concept of **SAM** (*Sharpness-Aware Minimization*) [28], which focuses on the geometry of the loss landscape, attempting to find neighborhoods of uniformly low loss. This new min-max optimization can be computed using gradient descent, and outperforms other methods in some tasks, especially in the context of noisy labels.

Intuitively, to minimize the loss of all values in a neighborhood, we set our objective over the loss within ϵ of our weight and use an ℓ_2 regularizer, such that our objective is over:

$$\min_{\theta} L_S^{SAM}(\theta) + \lambda \|\theta\|_2^2 \quad \text{where} \quad L_S^{SAM}(\theta) = \max_{\|\epsilon\|_p \leq \rho} L_S(\theta + \epsilon)$$

In solving the problem, the dual norm solution yields:

$$\hat{\epsilon}(\theta) \leftarrow \frac{\rho \cdot \text{sign}(\nabla_\theta L_S(\theta_{t-1})) \cdot |\nabla_\theta L_S(\theta_{t-1})|}{(\|\nabla_\theta L_S(\theta_{t-1})\|_q^q)^{1/p}}$$

and an approximation of $\nabla_\theta L_S^{SAM}(\theta)$ yields:

$$\nabla_\theta L_S^{SAM}(\theta) \approx \nabla_\theta L_S(\theta)|_{\theta + \hat{\epsilon}(\theta)}$$

Algorithm 11: SAM

Data: training set $\mathcal{S} = \bigcup_{n=1}^{N} \{(x_n, y_n)\}$, loss function $L_S(w) = \frac{1}{n} \sum_{i=1}^{n} \ell(w, x_i, y_i)$ where $\ell : \mathcal{W} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_+$, step size $\alpha > 0$, neighborhood size $\rho > 0$
Result: θ_t

$\theta_0 \leftarrow 0$
$t \leftarrow 0$
while θ_t *is not converged* **do**
  $t \leftarrow t + 1$
  $B \leftarrow \{(x_1, y_1), \ldots, (x_b, y_b)\}$ `/* Get sample batch */`
  $g_{t-1} \leftarrow \nabla_\theta f_B(\theta_{t-1})$ `/* Compute gradient on batch training loss */`
  $\hat{\epsilon}(\theta) \leftarrow \rho \cdot \text{sign}(\nabla_\theta L_S(\theta_{t-1})) \cdot \frac{|\nabla_\theta L_S(\theta_{t-1})|}{(\|\nabla_\theta L_S(\theta_{t-1})\|_q^q)^{1/p}}$ `/* Compute noise */`
  $g \leftarrow \nabla_\theta L_B(\theta_{t-1})|_{\theta + \hat{\epsilon}(\theta)}$ `/* Gradient approximation for SAM objective */`
  $\theta_t \leftarrow \theta_{t-1} - \alpha g$ `/* Descent using unit gradient */`
end
return θ_t

In 2021, Kwon et al. formulated a variation of SAM called **ASAM** [29], short for *Adaptive Sharpness-Aware Minimization*, to counter the sensitivity of sharpness when the parameter is re-scaled . They defined for a general family

of invertible linear operators $\{T_\theta, \theta \in \mathbb{R}^k\}$, T_θ^{-1} is the normalization operator of θ. Then in Algorithm 11, we remove lines 6 and 8, and define the following line for ϵ:

$$\epsilon \leftarrow \rho \cdot \frac{T_\theta^2 \nabla_\theta L_B(\theta_{t-1})}{\|T_\theta^2 \nabla_\theta L_B(\theta_{t-1})\|_2}$$

as well as the weight update step:

$$\theta_t \leftarrow \theta_{t-1} - \alpha(\nabla_\theta L_B(\theta_{t-1} + \epsilon) + \lambda\theta_{t-1})$$

In 2022, Du et al. then introduced another variation of SAM called **ESAM** [30], short for *Efficient Sharpness-Aware Minimization* to address the high computational cost of SAM . They proposed Stochastic Weight Perturbation (SWP), which approximated sharpness by perturbing stochastically chosen weights in each iteration, as well as Sharpness-Sensitive Data Selection (SDS), which optimized loss over a specialized subset of data sensitive to sharpness.

SWP chooses a random subset of weights from the original set of weights to perform backpropagation with, defines the perturbation as $\epsilon = \frac{\rho}{\beta}\nabla_\theta L_S(f_\theta)$ to ensure expected weight perturbation equal to $\hat{\epsilon}$. SDS splits the mini-batch B such that $B^+ = \{(x_i, y_i) \in B : \ell(f_{\theta+\hat{\epsilon}}, x_i, y_i) - \ell(f_\theta, x_i, y_i) > \alpha\}$ for some α, and defines the selection ratio γ as $|B^+|/|B|$.

Algorithm 12: ESAM

Data: training set $\mathcal{S} = \bigcup_{n=1}^{N}\{(x_n, y_n)\}$, loss function $L_S(w) = \frac{1}{n}\sum_{i=1}^{n}\ell(w, x_i, y_i)$ where $\ell : \mathcal{W} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_+$, step size $\alpha > 0$, neighborhood size $\rho > 0$, iterations A, SWP hyperparameter β, SDS hyperparameter γ

Result: $\hat{\theta}$

1 **for** $a = 1$ **to** A **do**
2 $B \leftarrow \{(x_1, y_1), \ldots, (x_b, y_b)\}$ `/* Get sample batch */`
3 **for** $n = 1$ **to** N **do**
4 $\epsilon_n \leftarrow 0$
5 **if** θ_n *chosen by probability* β **then**
6 $\epsilon_n \leftarrow \frac{\rho}{\beta}\nabla_\theta L_B(f_\theta)$ `/* Apply SWP */`
7 **end**
8 $\hat{\epsilon} \leftarrow (\epsilon_1, \ldots, \epsilon_N)$ `/* Weight perturbation */`
9 $B^+ \leftarrow$ SDS construction with selection ratio γ
10 $g \leftarrow \nabla_\theta L_{B^+}(f_{\theta+\hat{\epsilon}})$
11 $\theta \leftarrow \theta - \alpha \cdot g$
12 **end**
13 **end**
14 **return** θ

7.5 Perturbated Unit Gradient Descent (PUGD)

In 2021, Ching-Hsun et al. introduced **Perturbated Unit Gradient Descent** [31], which uses Sharpness-Awareness Minimization as well as locally bounded updating. This results in flatter minima since perturbation adds noise to find smooth loss.

In their paper, the authors first introduced a definition for tensor normalization, called the dual-norm. For $\|\cdot\|_p$ on a given tensor in $\mathbb{R}^{M\times I\times J}$, we can denote each $U_m \in \mathbb{R}^{I_{(m)}\times J_{(m)}}$ composed of elements $u^p_{i_{(m)},j_{(m)}}$ along row $i_{(m)}$, column $j_{(m)}$ and powered by p in a matrix. Then the dual-norm of the L_P-norm $\|U\|_p$ is defined where $\frac{1}{p} + \frac{1}{q} = 1$,

$$(\|U\|_p^p)^{\frac{1}{q}} = \left[\sum_{m=1}^{M}\left[\left[\left[\sum_{j=1}^{J_{(m)}}\sum_{i=1}^{I_{(m)}}\begin{bmatrix} u^p_{i_{(M)},j_{(M)}} & \cdots & u^p_{i_{(M)},J_{(M)}} \\ \vdots & \ddots & \vdots \\ u^p_{I_{(M)},j_{(M)}} & \cdots & u^p_{I_{(M)},J_{(M)}}\end{bmatrix}\right]^{\frac{1}{p}}\right]^{q}\right]^{\frac{1}{q}} \right.$$
$$\left. \vdots \right.$$
$$\left[\left[\left[\sum_{j=1}^{J_{(M)}}\sum_{i=1}^{I_{(M)}}\begin{bmatrix} u^p_{i_{(M)},j_{(M)}} & u^p_{i_{(M)},J_{(M)}} \\ u^p_{I_{(M)},j_{(M)}} & u^p_{I_{(M)},J_{(M)}}\end{bmatrix}\right]^{\frac{1}{p}}\right]^{q}\right] = \sup_{\|E\|\le 1} U^T V \quad \forall U, V \in \mathbb{R}^{M\times I\times J}$$

We then apply the definition in the PUGD algorithm using the dual norm when $p = q = 2$, and denote this using $\|\cdot\|$. Intuitively, the norm of the tensor uses the ℓ_2 norm of the inner matrices as the value to compute the outer norm.

Using this definition, we then define the unit tensor of tensor v as $\hat{v} = \frac{v}{\|v\|}$. We can then apply this definition to define the update step of **Unit Gradient Descent**:

$$\theta_t = \theta_{t-1} - \alpha_{t-1} \cdot \frac{g_{t-1}}{\|g_{t-1}\|}$$

where $\|g_{t-1}\|$ is the dual-norm of g_{t-1}.

Unit perturbation is then added to UGD to apply an adaptive mechanism using ASAM in a unit radius and locally bounded updates in the unit ball. The dual-norm also keeps the gradients non-zero, which prevents gradients from bouncing up. This produces the following algorithm:

Algorithm 13: Perturbated Unit Gradient Descent

Data: training set $\mathcal{S} = \bigcup_{n=1}^{N}\{(x_n, y_n)\}$, loss function $f_s(\theta) = \mathbb{E}[\ell(w, x_i, y_i)]$ for ith batch data and loss $\ell(w, x_i, y_i)$, step size α_t
Result: θ_t

$\theta_0 \leftarrow$ random initial weight
$t \leftarrow 0$
while θ_t *is not converged* **do**
  for i **in** *random sample batch* $B : \{(x_1, y_1), \ldots, (x_n, y_n)\}$ **do**
    $t \leftarrow t + 1$
    $g_{t-1} \leftarrow \nabla_\theta f_s(\theta_{t-1})$ `/* Compute gradient from batch set` s `*/`
    $\hat{\epsilon}_{t-1} \leftarrow \frac{|w_{t-1}| \cdot g_{t-1}}{\||w_{t-1}| \cdot g_{t-1}\|}$ `/* Compute noise */`
    $\theta_{(t-1)^*} \leftarrow \theta_{t-1} + \hat{\epsilon}_{t-1}$ `/* Perturb weight */`
    $g_{(t-1)^*} \leftarrow \nabla_\theta f_s(\theta_{t-1} + \hat{\epsilon}_{t-1})$ `/* Perturbation gradients */`
    $\theta_{t-1} \leftarrow \theta_{(t-1)^*} - \hat{\epsilon}_{t-1}$
    $U_{t-1} \leftarrow \frac{g_{(t-1)^*} + g_{t-1}}{\|g_{(t-1)^*} + g_t\|}$ `/* Unit gradients */`
    $\theta_t \leftarrow \theta_{t-1} - \alpha_{t-1} U_{t-1}$ `/* Descent using unit gradient */`
  end
end
return θ_t

8 Discussion

The process we took was to read numerous papers and summarize how they worked and what progress they made in the optimization theory research field. We then classified each paper by time and type of optimization, and produce a succinct summary paper. Additionally, inspired by Yang et al. [32], we produce an evolutionary tree summarizing the evolution of all the optimization algorithms we have surveyed so far in Figure 1.

Some gaps in our process involve how we found papers. Though we found many papers, we do not guarantee that we found all the latest and most impactful optimization papers. Additionally, because our initial inspiration was Lion, an optimizer derived from AdamW, our process mostly involved researching adaptive methods. Therefore, we are very likely to have missed optimizers that are variations of SGD.

Overall, our work indicates promising directions in both new variations of Adam that have vastly improved upon Adam, as well as new directions like SAM and LARS that optimize over a neighborhood in the loss landscape for better generalization, as well as fits optimization methods to specific neural net architectures.

Empirically, our results can be tied together by training on models over the same dataset using each optimization method. However, due to time limit, we were unable to do so. Instead, we suppose that a machine learning engineer may use our work to understand which optimization method they wish to use for their model. For example, if they care a lot about weight decay and want to use momentum, they may opt for AdamW or Lion. Alternatively, if the problem is in non-convex setting, they may choose to use Yogi, or if they plan to train some Transformer-based LLMs, RAdam might be a good option to consider.

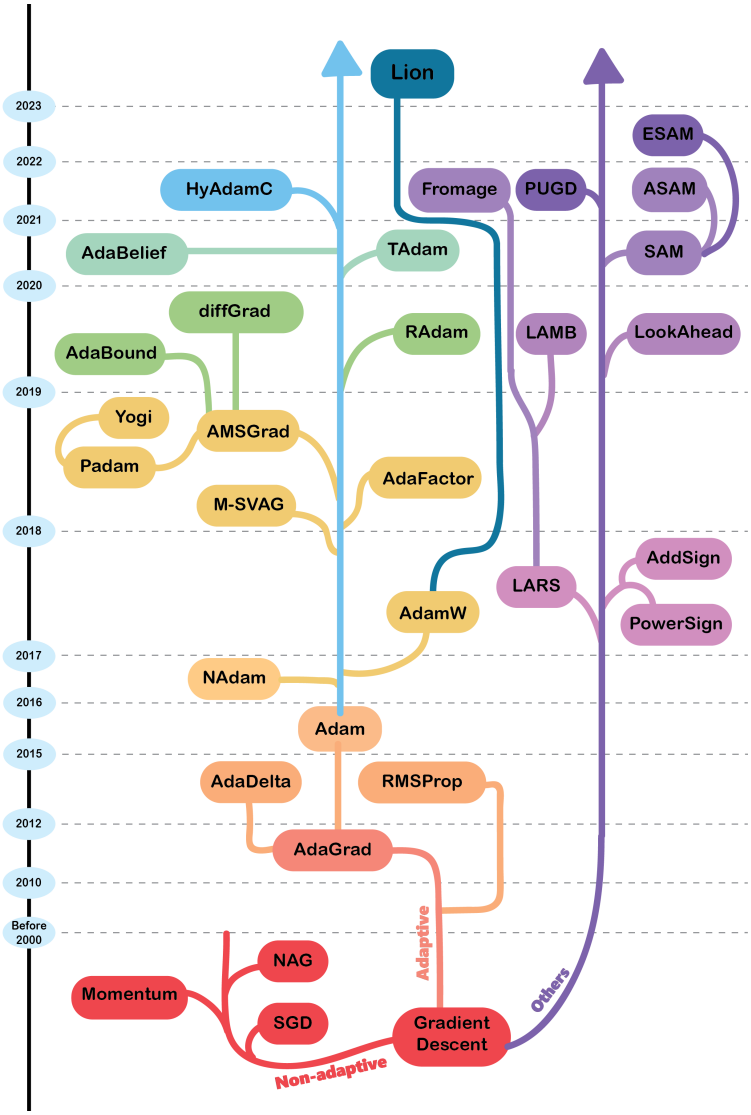

Figure 1: *The evolutionary tree of optimization algorithms. Models on the same branch have closer relationships.*

8.1 What was Easy

We found that it was straightforward to find many new optimization papers because of the vast quantity of them, and how each paper usually built off of another paper. There were many resources that we could use to find papers, including PapersWithCode and the Related Works and Experiment setup sections of each paper. Many papers also have their pseudocode ready and we can just adapt from there.

Additionally, we found it straightforward to classify each paper:

- 20th century papers were generally pre-adaptive.
- By the start of 2010s, there were notable adaptive papers that led to the introduction of Adam.
- Since Adam is so widely used, and that so many later optimizers clearly derived from Adam, we specifically have a separate section for variations of Adam.
- Finally, methods that applied different concepts from SGD and Adam were placed in the *Others* section.

8.2 What was Difficult

Reading through all 30 papers took more time than we expected. Because of the sheer amount text to go through, in addition to some papers requiring advanced knowledge of optimization theory, we had to read selectively to get the main contributions and algorithm differences. Additionally, while most papers have pseudocode included, their naming and variable conventions are not consistent, so we needed to pay extra attention to each paper's naming convention in order to make sure what we summarized above was consistent, especially for the variations of Adam.

8.3 Recommendations for Future Work

For a similar project that replicates or builds upon our work, we would recommend exploring more recent papers in more details, especially the branch of variations of SGD. There are a select number of well-known algorithms from before Adam. The more interesting and possibly more relevant papers today derive from the numerous variations of Adam, as well as the new directions of other optimization papers. For instance, new methods like SAM and PUGD redefine the optimization problem over a smooth, flat loss optimum, and LARS and its variations are empirically-motivated by optimization on specific neural network architectures. We would also suggest conducting an empirical study of optimizers' training convergence, loss, and generalization of some specific optimizers over a dataset such as CIFAR-100 or TinyImageNet to see if they actually improve compared their predecessors.

References

[1] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[2] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. Symbolic Discovery of Optimization Algorithms. *ArXiv*, abs/2302.06675, 2023.

[3] B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[4] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o(1/k^2). 1983.

[5] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[7] Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. 2012.

[8] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5 - RMSProp, COURSERA: Neural Networks for Machine Learning. 2012.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, arxiv.org/abs/1412.6980, 2014.

[10] Timothy Dozat. Incorporating Nesterov Momentum into Adam. In *Proceedings of the 4th International Conference on Learning Representations*, pages 1–4.

[11] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. 2019.

[12] Lukas Balles and Philipp Hennig. Dissecting Adam: The Sign, Magnitude and Variance of Stochastic Gradients, 2020.

[13] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive Learning Rates with Sublinear Memory Cost. 2018.

[14] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the Convergence of Adam and Beyond, 2019.

[15] Jinghui Chen, Dongruo Zhou, Yiqi Tang, Ziyan Yang, Yuan Cao, and Quanquan Gu. Closing the Generalization Gap of Adaptive Gradient Methods in Training Deep Neural Networks. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 3267–3275. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.

[16] Manzil Zaheer, Sashank J. Reddi, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. Adaptive Methods for Nonconvex Optimization. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 9815–9825, Red Hook, NY, USA, 2018. Curran Associates Inc.

[17] Liangchen Luo, Yuanhao Xiong, Yan Liu, and Xu Sun. Adaptive Gradient Methods with Dynamic Bound of Learning Rate. *CoRR*, abs/1902.09843, 2019.

[18] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019.

[19] Shiv Ram Dubey, Soumendu Chakraborty, Swalpa Kumar Roy, Snehasis Mukherjee, Satish Kumar Singh, and Bidyut Baran Chaudhuri. diffGrad: An Optimization Method for Convolutional Neural Networks, 2021.

[20] Wendyam Eric Lionel Ilboudo, Taisuke Kobayashi, and Kenji Sugimoto. TAdam: A Robust Stochastic Gradient Optimizer, 2020.

[21] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James S. Duncan. AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients, 2020.

[22] Kyung-Soo Kim and Yong-Suk Choi. HyAdamC: A new Adam-Based hybrid optimization algorithm for convolution neural networks. *Sensors (Basel)*, 21(12), June 2021.

[23] Yang You, Igor Gitman, and Boris Ginsburg. Large Batch Training of Convolutional Networks, 2017.

[24] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes, 2020.

[25] Jeremy Bernstein, Arash Vahdat, Yisong Yue, and Ming-Yu Liu. On the distance between two neural networks and the stability of learning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 21370–21381. Curran Associates, Inc., 2020.

[26] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Neural Optimizer Search with Reinforcement Learning. *ArXiv*, abs/1709.07417, 2017.

[27] Michael R. Zhang, James Lucas, Geoffrey Hinton, and Jimmy Ba. *Lookahead Optimizer: K Steps Forward, 1 Step Back*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[28] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-Aware Minimization for Efficiently Improving Generalization. *ArXiv*, abs/2010.01412, 2020.

[29] Jungmin Kwon, Jeongseop Kim, Hyunseong Park, and Inae Choi. ASAM: Adaptive Sharpness-Aware Minimization for Scale-Invariant Learning of Deep Neural Networks. In *International Conference on Machine Learning*, 2021.

[30] Jiawei Du, Hanshu Yan, Jiashi Feng, Joey Tianyi Zhou, Liangli Zhen, Rick Siow Mong Goh, and Vincent Y. F. Tan. Efficient Sharpness-aware Minimization for Improved Training of Neural Networks, 2022.

[31] Ching-Hsun Tseng, Liu Cheng, Shin-Jye Lee, and Xiaojun Zeng. Perturbed Gradients Updating within Unit Space for Deep Learning. *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 01–08, 2021.

[32] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, and Xia Hu. Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond, 2023.