An Implementation of Self-Tuning Stochastic Optimization with Curvature-Aware Gradient Filtering in PyTorch

Yash Patel
Department of Statistics
University of Michigan
Ann Arbor, MI 48104
yppatel@umich.edu

Abstract

Machine learning algorithms, especially in the very high dimensional regimes that are de rigueur nowadays, tend to be sensitive to choice of hyperparameters, such as that of the learning rate. As a result, many techniques have cropped up for adapting the learning rate over the course of training, but such methods tend to be not well justified theoretically. Here, we propose implementing an algorithm that self-tunes the learning rate based on a Kalman Filter denoised estimate of the gradient. The implementation is in the form of a PyTorch package with accompanying documentation, unit testing, code samples, and feature requests (following the convention of open source projects). The final implementation of the project can be seen at:

Repo: `https://github.com/yashpatel5400/CurveTorch`

Website: `https://curvetorch.herokuapp.com/`

Tutorials: `https://curvetorch.herokuapp.com/tutorials/`

Feature Requests: `https://github.com/yashpatel5400/CurveTorch/issues`

Preprint. Under review.

1 Background

Modern machine learning algorithms find themselves in parameter regimes where the only feasible method of optimization is some variant of gradient descent. Many variants of traditional gradient descent have popped up over the intervening years, such as those in [1] [2]. However, these algorithms all fundamentally hinge upon the taking noisy estimates of the "true" gradient using some mini-batch estimate, perhaps coupled with momentum dynamics in some optimizers. In turn, these optimizers tend to be extremely sensitive to the user's choice of hyperparameters, such as that of their learning rate. In particular, oftentimes to combat this sensitivity, users are forced to introduce ad-hoc learning rate annealing schedules, which can take the form of more primitive adaption schedules to more complex ones [3]. Despite its pervasive use, as with many aspects of modern machine learning, its theoretical basis is largely non-existent, with research still ongoing in the space [4]. As a result of such ad-hoc methods, there is no manner of knowing a priori what hyperparameters will work best for a given practitioner's problem at hand, oftentimes forcing them to simply use the default parameter values.

Thanks to advancements in autograd technology [5] [6], further avenues of exploration, including those accessing Hessian-based information, are now feasible. A recent paper by the name of "Self-Tuning Stochastic Optimization with Curvature-Aware" [7] has emerged in this space, seeking to modify the standard gradient descent optimizer to be able to self-tune its learning rate using a Kalman filter to get a "denoised" estimate of the gradient. The authors of that paper did not release their code, so the contributions here were:

- Implementation of their algorithm in PyTorch
- Corresponding examples of the optimizer in Jupyter notebooks
- Documentation in an accompanying website to the repository
- Exploration to momentum based algorithms

2 Model Recap

For the sake of completeness, we summarize the main findings of [7] here. To ease discussions between the two, we also adopt their choice of notation. As usual, the problem formulation is

$$\arg\min_{\theta \in \mathbb{R}^d} f(\theta), \quad f(\theta) = \mathbb{E}_\xi[\tilde{f}(\theta, \xi)], \tag{1}$$

where this ξ is the random variable underlying the observed sample $\{\xi^{(i)}\}$. In order to optimize the above function, we oftentimes use mini-batch gradient descent, updating the parameters by updating $\theta_{t+1} \leftarrow \theta_t - \alpha_t g_t$, where

$$g_t = \left(\frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \tilde{f}(\theta_t, \xi_t^{(i)}) \right). \tag{2}$$

Here, $\xi_t^{(i)}$ are the samples drawn for time step t. As shorthand, denote $f(\theta_t)$ as f_t and similarly $\nabla_\theta f(\theta_t)$ as ∇f_t. As noted above, one of the main issue hampering the ability to do auto-selection of hyperparameters is the use of the noisy gradient estimates from the mini-batch estimates. To combat this, an effective viewpoint to adopt is seeing these mini-batch estimates as noisy measurements of a true latent state of a dynamical system. As a result, the observations and dynamics of the system need to be explicitly modelled.

As with any latent model, we model $\mathbb{P}[g_t \mid \nabla f_t]$, which we assume follows a Gaussian as $g_t \mid \nabla f_t \sim \mathcal{N}(\nabla f_t, \Sigma_t)$. This model is justified by the fact that g_t are sample averages of the ∇f_t, meaning this follows by the Central Limit Theorem. For the dynamics, we wish to model the evolution of ∇f_t, which we do with $\mathbb{P}[\nabla f_t \mid \nabla f_{t-1}]$. The assumed dynamics arises from Taylor expansion of the gradient, namely $\nabla f(\theta_{t-1}) \approx \nabla f(\theta_{t-1}) - \nabla^2 f(\theta_t)\delta_{t-1}$, and results in finding

$$\nabla f_t \mid \nabla f_{t-1} \sim \mathcal{N}(\nabla f_{t-1} + B_t \delta_{t-1}, Q_t), \tag{3}$$

where $\mathbb{E}[B_t] = \nabla^2 f(\theta_t)$ and Q_t is the covariance of $B_t \delta_{t-1}$. The final estimators of this latent gradient, therefore, can be modelled and then iteratively updated with a Kalman Filter, defining the parameters of interest as m_t^-, m_t, P_t^-, P_t:

$$\nabla f_t \mid g_{1:t-1} \sim \mathcal{N}(m_t^-, P_t^-) \tag{4}$$
$$\nabla f_t \mid g_{1:t} \sim \mathcal{N}(m_t, P_t). \tag{5}$$

The dynamics of the update from a prior of $\nabla f_0 \sim \mathcal{N}(m_0, P_0)$ results in:

$$m_t^- = m_{t-1} + B_t \delta_{t-1}, \quad P_t^- = P_{t-1} + Q_{t-1} \tag{6}$$
$$K_t = P_t^- (P_t^- + \Sigma_t)^{-1} \tag{7}$$
$$m_t = (I - K_t) m_t^- + K_t g_t, \quad P_t = (I - K_t) P_t^- (I - K_t)^T + K_t \Sigma_t K_t^T \tag{8}$$

This is the key algorithm that ulimately produces the denoised gradient estimators that we use to self-tune the learning rate. In addition to the denoised gradient estimate, we need to model the dynamics of the function value itself, which we do similarly:

$$f_t \mid f_{t-1} \sim \mathcal{N}(f_{t-1} + m_{t-1}^T \delta_{t-1} + \frac{1}{2} \delta_{t-1}^T B_t \delta_{t-1}, \lambda_t + \delta_{t-1}^T P_{t-1} \delta_{t-1} + \frac{1}{4} \delta_{t-1}^T Q_t \delta_{t-1}) \tag{9}$$
$$y_t \mid f_t \sim \mathcal{N}(f_t, r_t) \tag{10}$$

From here, a similar set of Kalman Filter equations can be derived. In the original paper, this was not explicitly given, so we provide it as a concrete contribution to ease future implementation of the algorithm:

$$u_t^- = u_{t-1} + m_{t-1}^T \delta_{t-1} + \frac{1}{2} \delta_{t-1}^T B_t, \quad \lambda_t = \max\{(y_t - u_t^-)^2 - c_t, 0\} \tag{11}$$
$$s_t^- = \lambda_t + \underbrace{s_{t-1} + \delta_{t-1}^T P_{t-1} \delta_{t-1} + \frac{1}{4} \delta_{t-1}^T Q_t \delta_{t-1})}_{c_t}, \quad \gamma_t = \frac{s_t^-}{s_t^- + r_t} \tag{12}$$
$$u_t = (1 - \gamma_t) u_t^- + \gamma_t y_t, \quad s_t = (1 - \gamma_t)^2 s_t^- + \gamma_t^2 r_t \tag{13}$$

From these two, we define an optimization problem that gives us the "optimal" learning rate, where optimal is defined by the quantity on the right-hand side, representing the probability of improvement.

$$\alpha_{PI} := \arg\max_{\alpha} \mathbb{P}\left[f_{t+1} - f_t \leq 0 \mid y_{1:t}, g_{1:t}\right]. \tag{14}$$

It turns out that finding such an α_{PI} immediately follows from the derived Kalman Filter dynamics, by finding

$$\alpha_{PI} := \arg\min_{\alpha} \frac{-\alpha \delta_t^T m_t + \frac{\alpha^2}{2} \delta_t^T B_t \delta_t}{\sqrt{2 s_t + \alpha^2 \delta_t^T P_t \delta_t + \frac{\alpha^2}{4} \delta_t^T Q_t \delta_t}}. \tag{15}$$

This problem can quite efficiently be optimized using Newton's method, since it is only over a single variable with all the other variables fixed. The authors of the paper called this algorithm "MEKA." Combining these components gives the algorithm that was implemented in the accompanying PyTorch package.

3 Algorithm

Here, we summarize the algorithm that captures the derivations provided in the previous section. We also detail the important implementation details that were necessary to get the algorithm functioning in the following section. The algorithm can be summarized as follows:

Algorithm 1 MEKA optimization algorithm

Hyperparameters decay rates $\beta_r = .999, \beta_\Sigma = .999, \beta_\alpha = .999$
$m_0, P_0 \leftarrow \overrightarrow{0}, 10^4$
$u_0, s_0 \leftarrow 0, 10^4$
$\delta_0, s_0 \leftarrow \overrightarrow{0}$
$t \leftarrow 0$
while not converged **do**
  $t \leftarrow t + 1$
  $f_t^{(i)}, \nabla f_t^{(i)}, \nabla^2 f_t^{(i)} \delta_{t-1} \leftarrow \text{VectorizedMap}(f, \{x_i\}_{i=1}^M, \theta_{t-1})$
  $y_t, r_t \leftarrow \text{MeanVarEMA}(\{f_t^{(i)}, \beta_r\})$
  $g_t, \Sigma_t \leftarrow \text{MeanVarEMA}(\{\nabla f_t^{(i)}, \beta_r\})$
  $b_t, Q_t \leftarrow \text{MeanVar}(\{\nabla^2 f_t^{(i)} \delta_{t-1}\})$
  $u_t, s_t \leftarrow \text{FilterUpdate}(u_{t-1}, s_{t-1}, m_{t-1}, P_{t-1}, y_t, r_t, b_t, Q_t)$
  $m_t, P_t \leftarrow \text{FilterUpdate}(m_{t-1}, P_{t-1}, g_t, \Sigma_t, b_t, Q_t)$
  $\alpha_t \leftarrow \arg\min_\alpha 15$
  $\theta_t \leftarrow \theta_{t-1} - \delta_t$
end while

4 Implementation Details

To get this algorithm working, a number of practical modifications and concrete interpretations had to be made:

4.1 Variance Estimation

Part of the input of the Kalman Filter is an exponential moving variance of both the gradient function value and its associated Hessian vector product. In the derivations above, these were denoted Σ_t, Q_t respectively. As with the findings of the authors of the paper, we found that direct estimation of these matrices as covariances across the features led to instability in the algorithm convergence. Switching instead to a diagonal form with $\Sigma_t = \sigma_t I, Q_t = q_t I$, where each of the scalars corresponds to the averages across the diagonals of the original matrices, led to greater stability in the results. Further work may wish to explore the reason behind this instability of the covariance matrix.

4.2 Learning Rate Clamping

Recall that the learning rate is found as a solution to an optimization problem, namely 15, where the denominator is a function of P_t, Q_t, respectively the running averages of the function value and Hessian vector product. For the beginning of the optimization run, there is insufficient data collected on such values, which results in unrealistically low estimates of the variances, which in turn leads to degenerate forms of 15. As a result, the optimal learning rates turn out to be meaningless in such circumstances, tending to infinity. To combat this, we first clamp the variance estimates that are used to construct P_t, Q_t from below to be at least .01. This coupled with a direct clamping of the learning rate to some α_m generally resolves the issue of divergence.

5 Test Bed

To both confirm the correctness of our implementation and avoid regressions in contributions made by other researchers, a suite of unit tests were added. Each of these was selected to cover a spread of different function profiles, specifically from those referenced on [8]. These categories are: many local minima, bowl-shaped, plate-shaped, valley-shaped, and steep ridges/drops. The following test functions are present in the suite:

- **Quadratic (Bowl):** $x^2 + y^2$
- **Rosenbrock (Valley):** $(1-x)^2 + (1-x^2)^2$
- **Six-Hump Camel (Valley):** $4x^2 - 2.1x^4 + x^6/3 + xy - 4y^2 + 4y^4$
- **Ackley (Many Local Minima):** $-20e^{-.2\sqrt{.5(x^2+y^2)}} - e^{.5(\cos(2\pi x)+\cos(2\pi y))} + e + 20$
- **McCormick (Plate Shaped):** $\sin(x+y) + (x-y)^2 - 1.5x + 2.5y + 1$
- **Rastrigin (Many Local Minima):** $x^2 - \cos(2\pi x) + y^2 - \cos(2\pi y)$

6 Results

Using the implementation, the following results were obtained in the test bed functions. For all the graphs, the convention of using a gradient shaded trajectory is adopted, where the light tail of the line represents where the trajectory starts and the dark where it ends. Each graph is further captioned with the optimal point and corresponding value.

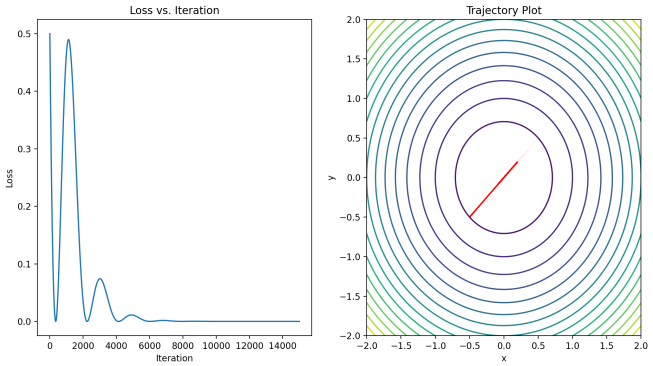

Figure 1: Quadratic optimization: $x^2 + y^2$. Known minimum: $(0, 0)$

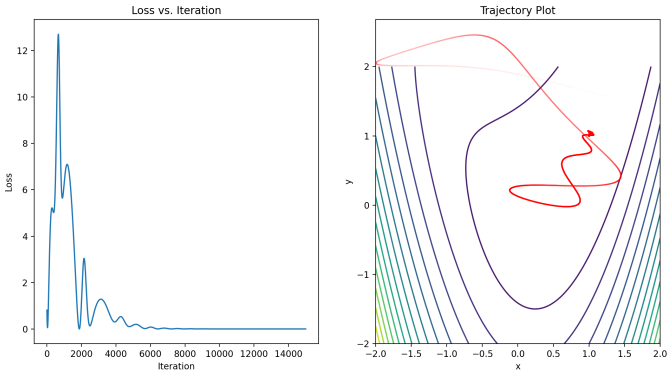

Figure 2: Rosenbrock optimization: $(1 - x)^2 + (1 - x^2)^2$. Known minimum: $(1, 1)$

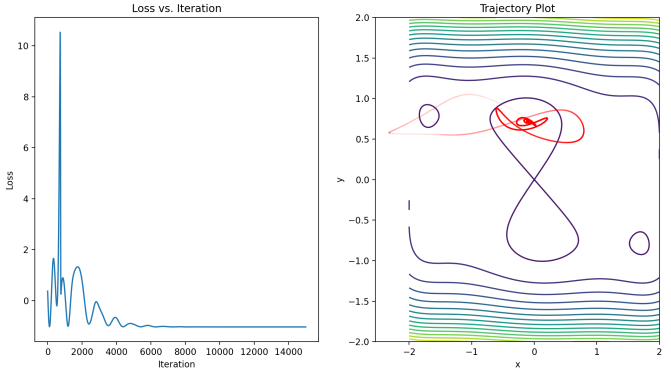

Figure 3: Six-hump camel optimization: $4x^2 - 2.1x^4 + x^6/3 + xy - 4y^2 + 4y^4$. Known minimum: $(-0.0898, 0.7126)$

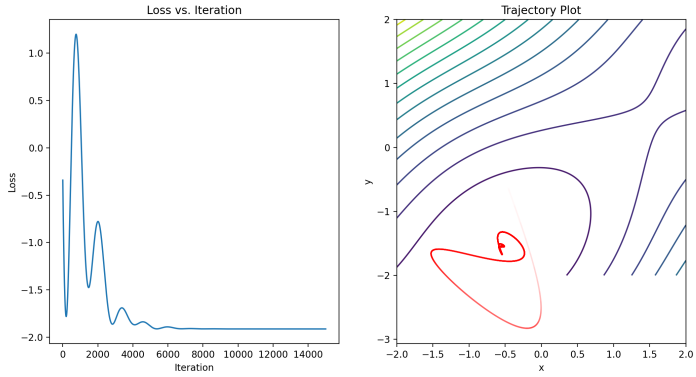

Figure 4: McCormick optimization: $\sin(x+y)+(x-y)^2-1.5x+2.5y+1$. Known minimum: $(-0.54719, -1.54719)$

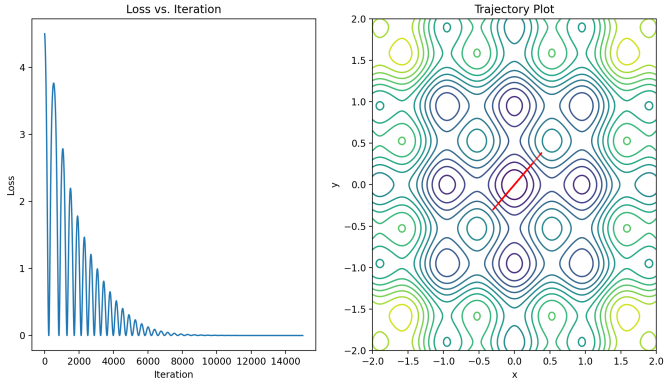

Figure 5: Rastrigin optimization: $x^2 - \cos(2\pi x) + y^2 - \cos(2\pi y)$. Known minimum: $(0, 0)$

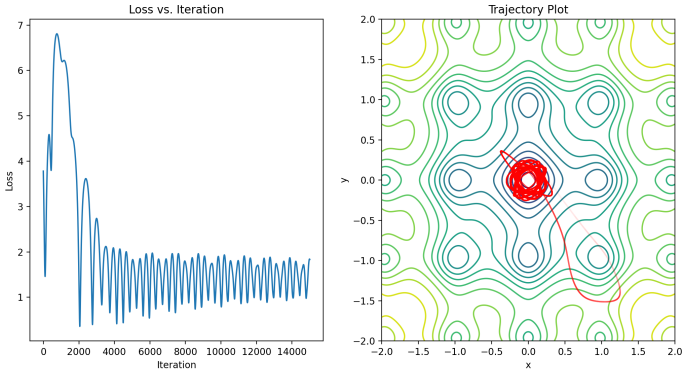

Figure 6: Ackley's optimization: $-20e^{-.2\sqrt{.5(x^2+y^2)}} - e^{.5(\cos(2\pi x)+\cos(2\pi y))} + e + 20$. Known minimum: $(0, 0)$

7 Discussion

From the above graphs, a number of notable trends stand out that are worth commenting on. The first is simply that, with the exception of the Ackley's function test, all the trials converged to their known minima. This gives confidence that the core PyTorch implementation from the paper is largely correct. This unlocks the ability for other experimenters to be able to build off of the base provided in the accompanying repo to either further explore this avenue of self corrected tuning or to simply make use of a subset of the implementation, such as the Kalman filtering of the gradient.

Returning to notable attributes of the results themselves, an interesting feature is that this optimizer is clearly **not** a strict descent optimizer, evident from the spikes in the loss functions in all the executions. This is clear from the original problem formulation in 15 seeking to maximize the **probability** of improvement. As a result, some of the steps should be expected to increase the loss function, especially initially, when the properties of the loss landscape have not been captured by the Kalman Filter latent states.

Despite robustness to most problems presented, we demonstrate that this algorithm fails to converge in the case of the Ackley's function, getting caught in a cycle around the minimum. Similar behavior was observed in the case of "many local minima" loss functions, with surrounding bumps having relatively high amplitudes. One potential explanation stems from the Kalman filtered gradient direction: getting caught in such "cycles" seems to prevent the Kalman filter from updating from its direction tangent to the cycle to the true direction perpendicular to it, likely due to insufficent weight. Further experiments of this technique in conjunction with momentum may reveal some feasible medium that allows such deficiencies to be overcome. Due to this deficiency though, some doubt arises about the practical utility of this particular algorithm for practical purposes, given that modern machine learning loss landscapes are plagued by local optima.

8 Conclusion

In this work, we have presented an implementation of "Self-Tuning Stochastic Optimization with Curvature-Aware Gradient Filtering" in PyTorch as a complete package, together with an automated testing suite, accompanying Jupyter notebook tutorials, documentation on an accompanying website, and requests for features to garner involvement from the broader community. In so doing, we affirmed properties initially documented in the paper, specifically about the general utility of this algorithm and further understood properties about its convergence properties across a suite of optimization problems. In so doing, deficiencies of the algorithm convergence in regimes of multiple high amplitude local optima became apparent, which reduces the general applicability of this algorithm to broader problems that may be of practical interest. Future work can include extensions that explore specifically the use of the Kalman filtered gradient in other contexts or the use of momentum to robustly improve the convergence properties of this algorithm.

References

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

[2] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[3] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 464–472. IEEE, 2017.

[4] Preetum Nakkiran. Learning rate annealing can provably help generalization, even for convex problems. *arXiv preprint arXiv:2005.07360*, 2020.

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[6] Ashish Agarwal and Igor Ganichev. Auto-vectorizing tensorflow graphs: Jacobians, auto-batching and beyond. *arXiv preprint arXiv:1903.04243*, 2019.

[7] Ricky TQ Chen, Dami Choi, Lukas Balles, David Duvenaud, and Philipp Hennig. Self-tuning stochastic optimization with curvature-aware gradient filtering. 2020.

[8] Sonja Surjanovic and Derek Bingham. Virtual library of simulation experiments: Test functions and datasets.