YellowFin and the Art of Momentum Tuning

Jian Zhang, Ioannis Mitliagkas, Christopher Ré
Department of Computer Science
Stanford University
`{zjian,imit,chrismre}@cs.stanford.edu`

June 13, 2017

Abstract

Hyperparameter tuning is one of the big costs of deep learning. State-of-the-art optimizers, such as Adagrad, RMSProp and Adam, make things easier by adaptively tuning an individual learning rate for each variable. This level of fine adaptation is understood to yield a more powerful method. However, our experiments, as well as recent theory by Wilson et al. [1], show that hand-tuned stochastic gradient descent (SGD) achieves better results, at the same rate or faster. The hypothesis put forth is that adaptive methods converge to different minima [1]. Here we point out another factor: none of these methods tune their momentum parameter, known to be very important for deep learning applications [2]. Tuning the momentum parameter becomes even more important in asynchronous-parallel systems: recent theory [3] shows that asynchrony introduces momentum-like dynamics, and that tuning down algorithmic momentum is important for efficient parallelization.

We revisit the simple momentum SGD algorithm and show that hand-tuning a single learning rate and momentum value makes it competitive with Adam. We then analyze its robustness in learning rate misspecification and objective curvature variation. Based on these insights, we design YellowFin, an automatic tuner for both momentum and learning rate in SGD. YellowFin optionally uses a novel momentum-sensing component along with a negative-feedback loop mechanism to compensate for the added dynamics of asynchrony on the fly. We empirically show YellowFin converges in fewer iterations than Adam on large ResNet and LSTM models, with a speedup up to 2.8x in synchronous and 2.7x in asynchronous settings.

1 Introduction

Accelerated forms of stochastic gradient descent (SGD), pioneered by Polyak [4] and Nesterov [5], are the de-facto training algorithms for deep learning. Their use requires a sane choice for their *hyperparameters*: typically a *learning rate* and *momentum parameter* [2]. However, tuning hyperparameters is arguably the most time-consuming part of deep learning, with thousands of productive hours sacrificed and many papers outlining best tuning practices written [6, 7, 8, 9].

Deep learning researchers have proposed a number of methods to deal with hyperparameter optimization. Naïve methods, like the grid-search, are prohibitively expensive for all but the smallest problems. Smart black-box methods [10, 11] do not explicitly take into account the problem specifics and spend time testing multiple configurations. Adaptive methods provide an attractive alternative. They aim to tune a single run on the fly and have been largely successful in relieving practitioners of tuning the learning rate. Algorithms like Adagrad [12], RMSProp [13] and Adam [14] use the magnitude of gradient elements to tune learning rates *individually for each variable*. This increased flexibility sounds great, however our experiments and recent analysis in literature [1] suggest that methods that adapt multiple learning rates, yield marginal benefits compared to momentum SGD. Wilson et al. [1] argue that those methods also have worse generalization. We make another argument: adaptive methods also suffer because they do not tune their momentum parameter.

Momentum is a fundamental parameter at the heart of accelerated optimization, lending its name to the most ubiquitous accelerated method [4], commonly referred to as *momentum*. Classic [4] and recent results [2] alike show that proper momentum tuning has a significant impact on training speed. Momentum becomes even more critical on distributed systems. Recently, Mitliagkas et al. [3] showed that, in asynchronous-parallel systems—a popular design for efficient distributed training without synchronization [15, 16, 17, 18]—the system introduces momentum-like dynamics into the optimization. As a result, one should reduce the amount of algorithmic momentum to get fast convergence [18]. As part of a collaboration with an industry affiliate and a big research lab, we were able to verify that tuning momentum improves convergence on thousand-node scales. However, grid-searching momentum on very large cluster jobs becomes especially challenging. Better understanding of momentum and its rich properties is interesting in its own right and could yield a next generation of adaptive methods that perform *automatic momentum tuning*.

We revisit the basic SGD update that uses Polyak's momentum and a single learning rate for all variables. We empirically show that, when hand-tuned, momentum SGD achieves faster convergence than Adam for a large class of models. We then formulate the optimization update as a dynamical system and study certain robustness properties of the momentum operator. Building on our analysis, we design YellowFin, an automatic hyperparameter tuner for momentum SGD. YellowFin tunes the learning rate and momentum on the fly, and uses a novel *closed-loop control architecture* to compensate for the added dynamics of asynchrony. Specifically:

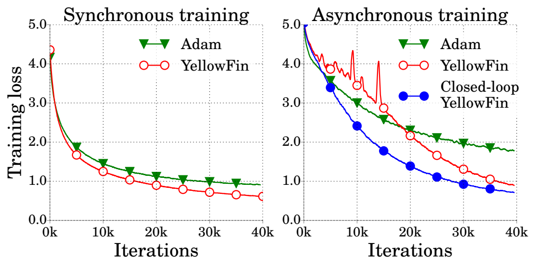

Figure 1: YellowFin in comparison to Adam on a ResNet (CIFAR100, cf. Section 5).

- Our analysis in Section 2 shows that momentum is robust to learning rate misspecification and curvature variation, two desirable properties for deep learning. These properties stem from a known but obscure fact: the momentum operator's spectral radius is constant in a large subset of the hyperparameter space.
- In Section 3, we use these robustness insights and a simple quadratic model analysis to design YellowFin, an automatic tuner for momentum SGD. YellowFin uses on-the-fly measurements from the gradients of the system to tune both learning rate and momentum.
- In Section 4 we present closed-loop YellowFin, suited for asynchronous training. It measures the total momentum in a running system, including any asynchrony-induced momentum. This measurement is used in a negative feedback loop to control the value of algorithmic momentum.

In Section 5, we demonstrate empirically that on ResNets and LSTMs YellowFin converges in fewer iterations compared to: (i) hand-tuned momentum SGD (a speedup of up to 2.2x); and (ii) hand-tuned Adam (up to 2.8x speedup). Under asynchrony, we demonstrate that state-of-the-art adaptive methods suffer due to lack of momentum tuning. In the same setting, the closed-loop control architecture speeds up YellowFin by up to 2x, making it at least 2.7x faster than Adam. We conclude with related work in Section 6 and discussion in Section 7. We release a TensorFlow implementation of YellowFin, that can be used as drop-in replacement for any optimizer[1]. We are also planning a PyTorch release.

2 The momentum operator

In this section we identify the main technical insights that guide the design of YellowFin. We first establish some preliminaries on momentum gradient descent and then perform a simple dynamical analysis. We show that momentum is robust to learning rate misspecification and curvature variation for a class of non-convex objectives, two important properties for deep learning.

2.1 Preliminaries

We aim to minimize some objective $f(x)$. In machine learning, x is referred to as *the model* and the objective is some *loss function.* A low loss implies a well-fit model. Gradient descent-based procedures use the gradient of the objective function, $\nabla f(x)$, to update the model iteratively. Polyak's momentum gradient descent update [4] is given by

$$x_{t+1} = x_t - \alpha \nabla f(x_t) + \mu(x_t - x_{t-1}), \tag{1}$$

where α denotes the learning rate and μ the value of momentum used. Momentum's main appeal is its established ability to *accelerate convergence* [4]. On a strongly convex smooth function with condition number κ, the optimal convergence rate of gradient descent ($\mu = 0$) is $O(\frac{\kappa-1}{\kappa+1})$ [19]. On the other hand, for certain classes of strongly convex and smooth functions, like quadratics, the optimal momentum value,

$$\mu^* = \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^2, \tag{2}$$

yields the optimal accelerated rate $O(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})$.[2] This is the smallest value of momentum that **ensures the same rate of convergence along all directions**. This fact is often hidden away in proofs. We shed light on some of its previously unknown implications in Section 2.2.

2.2 Robustness properties of the momentum operator

In this section we analyze the dynamics of momentum on a simple class of one dimensional, non-convex objectives. We first introduce the notion of *generalized curvature* and use it to describe the momentum operator. Then we discuss some properties of the momentum operator.

[1]Our TensorFlow implementation can be accessed at `https://github.com/JianGoForIt/YellowFin`.

[2]This guarantee does not generalize to arbitrary strongly convex functions [20]. Nonetheless, acceleration is typically observable in practice (cf. Section 2.2).

Definition 1 (Generalized curvature). *The derivative of $f(x) : \mathbb{R} \to \mathbb{R}$, can be written as*

$$f'(x) = h(x)(x - x^*) \tag{3}$$

for some $h(x) \in \mathbb{R}$, where x^* is the global minimum of $f(x)$. We call $h(x)$ the generalized curvature*.*

The generalized curvature describes, in some sense, curvature with respect to the optimum, x^*. For quadratic objectives, it coincides with the standard definition of curvature. It is the sole quantity related to the objective that influences the dynamics of gradient descent. For example, the contraction of a gradient descent step is $1 - \alpha h(x_t)$. Let $\boldsymbol{A}_t$ denote the *momentum operator* at time t. Using a state-space augmentation, we can express the momentum update as

$$\begin{pmatrix} x_{t+1} - x^* \\ x_t - x^* \end{pmatrix} = \begin{bmatrix} 1 - \alpha h(x_t) + \mu & -\mu \\ 1 & 0 \end{bmatrix} \begin{pmatrix} x_t - x^* \\ x_{t-1} - x^* \end{pmatrix} \triangleq \boldsymbol{A}_t \begin{pmatrix} x_t - x^* \\ x_{t-1} - x^* \end{pmatrix}. \tag{4}$$

Lemma 2 (Robustness of the momentum operator). *If the generalized curvature, h, and hyperparameters α, μ are in the* robust region*, that is:*

$$(1 - \sqrt{\mu})^2 \leq \alpha h(x_t) \leq (1 + \sqrt{\mu})^2, \tag{5}$$

then the spectral radius of the momentum operator only depends on momentum: $\rho(\boldsymbol{A}_t) = \sqrt{\mu}$.

We can explain Lemma 2 as robustness with respect to learning rate and to variations in curvature.

Momentum is robust to learning rate misspecification For a one dimensional strongly convex quadratic objective, we get $h(x) = h$ for all x and Lemma 2 suggests that $\rho(\boldsymbol{A}_t) = \sqrt{\mu}$ as long as

$$(1 - \sqrt{\mu})^2/h \leq \alpha \leq (1 + \sqrt{\mu})^2/h. \tag{6}$$

In Figure 2, we plot $\rho(\boldsymbol{A}_t)$ for different α and μ. As we increase the value of momentum, the optimal rate of convergence $\sqrt{\mu}$ is achieved by an ever-widening range of learning rates. Furthermore, for objectives with large condition number, higher values of momentum are *both faster and more robust*. **This property influences the design of our tuner:** as long as the learning rate satisfies (6), we are in the *robust region* and expect the same asymptotics. For example a convergence rate of $\sqrt{\mu}$ for quadratics, independent of the learning rate. Having established that, we can just focus on optimally tuning momentum.

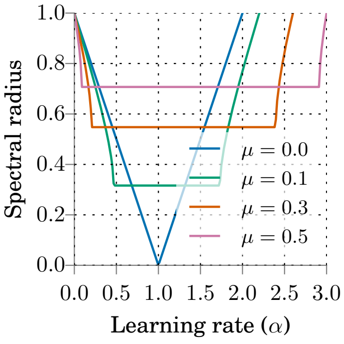

Figure 2: Spectral radius of momentum operator on scalar quadratic.

Momentum is robust to curvature variation As discussed in Section 2.1, the intuition hidden in classic results is that for strongly convex smooth objectives, the momentum value in (2) guarantees the same rate of convergence along all directions. We extend this intuition to certain non-convex functions. Lemma 2 guarantees a constant, time-homogeneous spectral radius for the momentum operators $(\boldsymbol{A}_t)_t$ if (5) is satisfied at every step. This motivates an extension of the condition number.

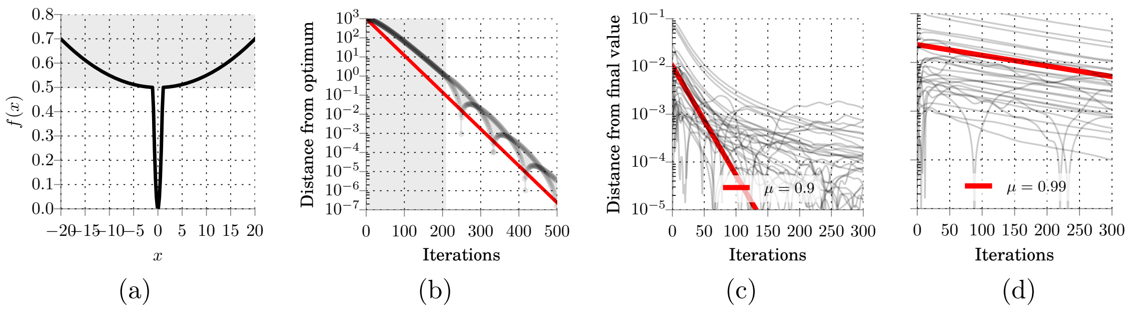

Figure 3: (a) Non-convex toy example; (b) constant convergence rate achieved empirically on the objective of (a) tuned according to (8); (c,d) LSTM on MNIST: as momentum increases, more variables (shown in grey) fall in the robust region and follow the robust rate, $\sqrt{\mu}$.

Definition 3 (Generalized condition number)**.** *We define the generalized condition number (GCN) of a scalar function, $f(x) : \mathbb{R} \to \mathbb{R}$, to be the dynamic range of its generalized curvature, $h(x)$:*

$$\nu = \frac{\sup_{x \in dom(f)} h(x)}{\inf_{x \in dom(f)} h(x)} \tag{7}$$

The GCN captures variations in generalized curvature along a scalar slice. From Lemma 2 we get

$$\mu^* = \left(\frac{\sqrt{\nu} - 1}{\sqrt{\nu} + 1} \right)^2, \quad \alpha^* = \frac{(1 - \sqrt{\mu})^2}{\inf_{x \in dom(f)} h(x)} \tag{8}$$

as the optimal hyperparameters. Specifically, μ^* is the smallest momentum value that guarantees a homogeneous spectral radius of $\sqrt{\mu^*}$ for all $(\boldsymbol{A}_t)_t$. The spectral radius of an operator describes its asymptotic behavior. However, the product of a sequence of operators $\boldsymbol{A}_t \cdots \boldsymbol{A}_1$ all with spectral radius $\sqrt{\mu}$ does not necessarily follow the asymptotics of $\sqrt{\mu}^t$. In other words, *we do not provide a convergence rate guarantee.* Instead, we provide empirical evidence in support of this intuition.

For example, the non-convex objective in Figure 3(a), composed of two quadratics with curvatures 1 and 1000, has a GCN of 1000. Using the tuning rule of (8), and running the momentum algorithm (Figure 3(b)) yields a practically constant rate of convergence throughout. In Figures 3(c,d) we demonstrate that for an LSTM, the majority of variables follows a $\sqrt{\mu}$ convergence rate. **This property influences the design of our tuner:** in the next section we use the tuning rules of (8) in YellowFin, generalized appropriately to handle SGD noise.

3 The YellowFin tuner

In this section we describe YellowFin, our tuner for momentum SGD. We introduce a noisy quadratic model and work on a local quadratic approximation of $f(x)$ to apply the tuning rule of (8) to SGD on an arbitrary objective. YellowFin is our implementation of that rule.

Noisy quadratic model We consider minimizing the one-dimensional quadratic

$$f(x) = \frac{1}{2}hx^2 + C = \frac{1}{n}\sum_i \frac{1}{2}h(x - c_i)^2 \triangleq \frac{1}{n}\sum_i f_i(x), \quad \sum_i c_i = 0. \tag{9}$$

The objective is defined as the average of n *component functions*, f_i. This is a common model for SGD, where we use only a single data point (or a mini-batch) drawn uniformly at random, $S_t \sim \mathrm{Uni}([n])$ to compute a noisy gradient, $\nabla f_{S_t}(x)$, for step t. Here, $C = \frac{1}{2n}\sum_i hc_i^2$ denotes the *gradient variance*. This scalar model is sufficient to study an arbitrary local quadratic approximation: optimization on quadratics decomposes trivially into scalar problems along the principal eigenvectors of the Hessian. Next we get an *exact* expression for the mean square error after running momentum SGD on a scalar quadratic for t steps.

Lemma 4. *Let $f(x)$ be defined as in* (9)*, $x_1 = x_0$ and x_t follow the momentum update* (1) *with stochastic gradients $\nabla f_{S_t}(x_{t-1})$ for $t \geq 2$. In expectation, squared distance to the optimum, x^*, is*

$$\mathbb{E}(x_{t+1} - x^*)^2 = (\boldsymbol{e}_1^\top \boldsymbol{A}^t [x_1 - x^*, x_0 - x^*]^\top)^2 + \alpha^2 C \boldsymbol{e}_1^\top (\boldsymbol{I} - \boldsymbol{B}^t)(\boldsymbol{I} - \boldsymbol{B})^{-1}\boldsymbol{e}_1, \tag{10}$$

where the first and second term correspond to squared bias and variance, and their corresponding momentum dynamics are captured by operators

$$\boldsymbol{A} = \begin{bmatrix} 1 - \alpha h + \mu & -\mu \\ 1 & 0 \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} (1 - \alpha h + \mu)^2 & \mu^2 & -2\mu(1 - \alpha h + \mu) \\ 1 & 0 & 0 \\ 1 - \alpha h + \mu & 0 & -\mu \end{bmatrix}. \tag{11}$$

Even though it is possible to numerically work on (10) directly, we use a scalar, asymptotic surrogate based on the spectral radii of operators to simplify analysis and expose insights. This decision is supported by our findings in Section 2: the spectral radii can capture empirical convergence speed.

$$\mathbb{E}(x_t - x^*)^2 \approx \rho(\boldsymbol{A})^{2t}(x_0 - x_*)^2 + (1 - \rho(\boldsymbol{B})^t)\frac{\alpha^2 C}{1 - \rho(\boldsymbol{B})} \tag{12}$$

One of our design decisions for YellowFin, is to always work in the robust region of Lemma 2. We know that this implies a spectral radius $\sqrt{\mu}$ of the momentum operator, $\boldsymbol{A}$, for the bias. Lemma 5 shows that under the exact same condition, the variance operator, $\boldsymbol{B}$ has spectral radius μ.

Lemma 5. *The spectral radius of the variance operator, $\boldsymbol{B}$ is μ, if $(1 - \sqrt{\mu})^2 \leq \alpha h \leq (1 + \sqrt{\mu})^2$.*

As a result, the surrogate objective of (12), takes the following form in the robust region.

$$\mathbb{E}(x_t - x^*)^2 \approx \mu^t (x_0 - x^*)^2 + (1 - \mu^t)\frac{\alpha^2 C}{1 - \mu} \tag{13}$$

We use this surrogate objective to extract a noisy tuning rule for YellowFin.

3.1 The tuner

Based on the surrogate in (13), we present YellowFin (Algorithm 1). Let D denote an estimate of the current model's distance to a local quadratic approximation's minimum, and C denote an estimate for gradient variance. Also, let h_{max} and h_{min} denote estimates for the largest and smallest generalized curvature respectively. The extremal curvatures h_{min} and h_{max} are meant to capture both curvature variation along different directions (like the classic condition number) and also variation that occurs as the *landscape evolves.* At each iteration, we solve the following SINGLESTEP problem.

(SINGLESTEP)

$$\begin{aligned} \mu_t, \alpha_t = \quad & \arg\min_{\mu} \mu D^2 + \alpha^2 C \\ s.t. \quad & \mu \geq \left(\frac{\sqrt{h_{\max}/h_{\min}} - 1}{\sqrt{h_{\max}/h_{\min}} + 1} \right)^2 \\ & \alpha = \frac{(1 - \sqrt{\mu})^2}{h_{\min}} \end{aligned}$$

Algorithm 1 YELLOWFIN

```
state: $\alpha \leftarrow 1.0$, $\mu \leftarrow 0.0$
function YELLOWFIN(gradient $g_t$, $\beta$)
    $h_{\max}, h_{\min} \leftarrow$ CURVATURERANGE($g_t, \beta$)
    $C \leftarrow$ VARIANCE($g_t, \beta$)
    $D \leftarrow$ DISTANCE($g_t, \beta$)
    $\mu_t, \alpha_t \leftarrow$ SINGLESTEP($C, D, h_{\max}, h_{\min}$)
    $\mu \leftarrow \beta \cdot \mu + (1 - \beta) \cdot \mu_t$
    $\alpha \leftarrow \beta \cdot \alpha + (1 - \beta) \cdot \alpha_t$
    return $\mu, \alpha$
end function
```

SINGLESTEP minimizes the surrogate for the expected squared distance from the optimum of a local quadratic approximation (13) after a single step ($t = 1$), while keeping all directions in the robust region (5). This is the SGD version of the noiseless tuning rule in (8). It can be solved in closed form; we refer to Appendix D for discussion on the closed form solution. YELLOWFIN uses functions CURVATURERANGE, VARIANCE and DISTANCE to measure quantities $h_{\max}$, $h_{\min}$, C and D respectively. These measurement functions can be designed and implemented in different ways. We present the implementations we used for our experiments, based completely on gradients, in Section 3.2.

3.2 Measurement functions in YELLOWFIN

This section describes our implementation of the measurement oracles used by YELLOWFIN: CURVATURERANGE, VARIANCE, and DISTANCE. We design the measurement functions with the assumption of a negative log-probability objective; this is in line with typical losses in machine learning, e.g. cross-entropy for neural nets and maximum likelihood estimation in general. Under this assumption, the Fisher information matrix—i.e. the expected outer product of noisy gradients—equals the Hessian of the objective [21, 22]. This allows for measurements purely from minibatch gradients with overhead linear to model dimensionality. These implementations are not guaranteed to give accurate measurements. Nonetheless, their use in our experiments in Section 5 shows that they are sufficient for YELLOWFIN to outperform the state of the art on a variety of objectives.

Algorithm 2 Curvature range

```
state: $h_{\max}$, $h_{\min}$, $h_i, \forall i \in \{1, 2, 3, ...\}$
function CURVATURERANGE(gradient $g_t$, $\beta$)
    $h_t \leftarrow \|g_t\|^2$
    $h_{\max,t} \leftarrow \max_{t-w \leq i \leq t} h_i$, $h_{\min,t} \leftarrow \min_{t-w \leq i \leq t} h_i$
    $h_{\max} \leftarrow \beta \cdot h_{\max} + (1 - \beta) \cdot h_{\max,t}$
    $h_{\min} \leftarrow \beta \cdot h_{\min} + (1 - \beta) \cdot h_{\min,t}$
    return $h_{\max}$, $h_{\min}$
end function
```

Algorithm 3 Gradient variance

```
state: $\overline{g^2} \leftarrow 0$, $\overline{g} \leftarrow 0$
function VARIANCE(gradient $g_t$, $\beta$)
    $\overline{g^2} \leftarrow \beta \cdot \overline{g^2} + (1 - \beta) \cdot g_t \odot g_t$
    $\overline{g} \leftarrow \beta \cdot \overline{g} + (1 - \beta) \cdot g_t$
    return $\|\overline{g^2} - \overline{g}^2\|_1$
end function
```

Algorithm 4 Distance to opt.

```
state: $\overline{\|g\|} \leftarrow 0$, $\overline{h} \leftarrow 0$
function DISTANCE(gradient $g_t$, $\beta$)
    $\overline{\|g\|} \leftarrow \beta \cdot \overline{\|g\|} + (1 - \beta) \cdot \|g_t\|$
    $\overline{h} \leftarrow \beta \cdot \overline{h} + (1 - \beta) \cdot \|g_t\|^2$
    $D \leftarrow \beta \cdot D + (1 - \beta) \cdot \overline{\|g\|}/\overline{h}$
    return $D$
end function
```

Curvature range Let g_t be a noisy gradient, we estimate the range of curvatures in Algorithm 2. We notice that the outer product of g_t has an eigenvalue $h_t = \|g_t\|^2$ with eigenvector g_t. Thus under our negative log-likelihood assumption, we use h_t to approximate the curvature of Hessian along gradient direction g_t. Specifically, we maintain $h_{\min}$ and $h_{\max}$ as running averages of extreme curvature $h_{\min,t}$ and $h_{\max,t}$, from a sliding window of width 20. As gradient directions evolve, we explore curvatures along different directions. Thus $h_{\min}$ and $h_{\max}$ capture the curvature variations.

Gradient variance To estimate the gradient variance in Algorithm 3, we use running averages $\overline{g}$ and $\overline{g^2}$ to keep track of g_t and $g_t \odot g_t$, the first and second order moment of the gradient. As $\mathrm{Var}(g_t) = \mathbb{E}g_t^2 - (\mathbb{E}g_t)^2$, we estimate the gradient variance C in (14) using $C = \|\overline{g^2} - \overline{g}^2\|_1$.

Distance to optimum In Algorithm 4, we estimate the distance to the optimum of the local quadratic approximation. Inspired by the fact that $\|\nabla f(\boldsymbol{x})\| \leq \|\boldsymbol{H}\|\|\boldsymbol{x} - \boldsymbol{x}^\star\|$ for a quadratic $f(x)$ with Hessian $\boldsymbol{H}$ and minimizer $\boldsymbol{x}^*$, we first maintain $\overline{h}$ and $\overline{\|g\|}$ as running averages of curvature h_t and gradient norm $\|g_t\|$. Then the distance is approximated using $\overline{\|g\|}/\overline{h}$.

4 Closed-loop YellowFin

To handle the momentum dynamics of asynchronous parallelism, we propose a *closed momentum loop* variant of YellowFin. After some preliminaries, we show the mechanism of the extension: it measures the dynamics on a running system and controls momentum with a negative feedback loop.

Preliminaries Asynchrony is a popular parallelization technique [15] that avoids synchronization barriers. When training on M asynchronous workers, staleness (the number of model updates between a worker's read and write operations) is on average $\tau = M - 1$, i.e., the gradient in the SGD update is delayed by τ iterations as $\nabla f_{S_{t-\tau}}(x_{t-\tau})$. Asynchrony yields faster steps, but can increase the number of iterations to achieve the same solution, a tradeoff between hardware and statistical efficiency [23]. Mitliagkas et al. [3] interpret asynchrony as added momentum dynamics. Experiments in Hadjis et al. [18] support this finding, and demonstrate that reducing algorithmic momentum can compensate for asynchrony-induced momentum and significantly reduce the number of iterations for convergence. Motivated by that result, we use the model in (14), where the total momentum, μ_T, includes both asynchrony-induced and algorithmic momentum, μ, in (1).

$$\mathbb{E}[x_{t+1} - x_t] = \mu_T \mathbb{E}[x_t - x_{t-1}] - \alpha \mathbb{E}\nabla f(x_t) \tag{14}$$

We will use this expression to design an estimator for the value of total momentum, $\hat{\mu_T}$. This estimator is a basic building block of closed-loop YellowFin, that *removes the need to manually compensate for the effects of asynchrony.*

Measuring the momentum dynamics Closed-loop YellowFin estimates total momentum μ_T on a running system and uses a negative feedback loop to adjust algorithmic momentum accordingly. Equation (14) gives an estimate of $\hat{\mu_T}$ on a system with staleness τ, based on (14).

$$\hat{\mu_T} = \mathsf{median}\left(\frac{x_{t-\tau} - x_{t-\tau-1} + \alpha \nabla_{S_{t-\tau-1}} f(x_{t-\tau-1})}{x_{t-\tau-1} - x_{t-\tau-2}}\right) \tag{15}$$

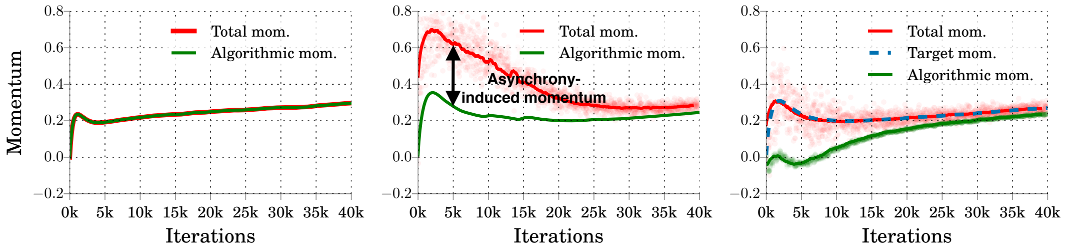

Figure 4: Momentum dynamics on CIFAR100 ResNet. Running YellowFin, total momentum is equal to algorithmic momentum in a synchronous setting (left). Total momentum is greater than algorithmic momentum on 16 asynchronous workers, due to asynchrony-induced momentum (middle). Using the momentum feedback mechanism of closed-loop YellowFin, lowers algorithmic momentum and brings total momentum to match the target value on 16 asynchronous workers (right). Red dots are individual total momentum estimates, $\hat{\mu}_T$, at each iteration. The solid red line is a running average of those estimates.

We use τ-stale model values to match the staleness of the gradient, and perform all operations in an elementwise fashion. This way we get a total momentum measurement from each variable; the median combines them into a more robust estimate.

Closing the asynchrony loop Given a reliable measurement of μ_T, we can use it to adjust the value of algorithmic momentum so that the total momentum matches the *target momentum* as decided by YellowFin in Algorithm 1. Closed-loop YellowFin in Algorithm 5 uses a simple negative feedback loop to achieve the adjustment. Figure 4 demonstrates that under asynchrony the measured total momentum is strictly higher than the algorithmic momentum (middle plot), as expected from theory; closing the feedback loop (right plot) leads to total momentum matching the target momentum. Closing the loop, as we will see, improves performance significantly. Note for asynchronous-parallel training, as the estimates and parameter tuning is unstable in the beginning when there are only a small number of iterations, we use initial learning $\frac{1}{\tau+1}$ instead of 1.0 to prevent overflow in the beginning.

Algorithm 5 Closed-loop YellowFin

```
1: Input: $\mu \leftarrow 0$, $\alpha \leftarrow \frac{1}{\tau+1}$, $\gamma \leftarrow 0.01, \tau$ (staleness)
2: for $t \leftarrow 1$ to $T$ do
3:     $x_t \leftarrow x_{t-1} + \mu(x_{t-1} - x_{t-2}) - \alpha \nabla_{S_t} f(x_{t-\tau-1})$
4:     $\mu^*, \alpha \leftarrow \text{YellowFin}(\nabla_{S_t} f(x_{t-\tau-1}), \beta)$
5:     $\hat{\mu_T} \leftarrow \text{median}\left(\frac{x_{t-\tau} - x_{t-\tau-1} + \alpha \nabla_{S_{t-\tau-1}} f(x_{t-\tau-1})}{x_{t-\tau-1} - x_{t-\tau-2}}\right)$      ▷ Measuring total momentum
6:     $\mu \leftarrow \mu + \gamma \cdot (\mu^* - \hat{\mu_T})$      ▷ Closing the loop
7: end for
```

5 Experiments

In this section, we empirically validate the importance of momentum tuning and evaluate YellowFin in both synchronous (single-node) and asynchronous settings. In synchronous settings, we first demonstrate that, with hand-tuning, momentum SGD is competitive with Adam, a state-of-the-art adaptive method. Then, we evaluate YellowFin without any hand tuning in comparison to hand-tuned Adam and momentum SGD. In asynchronous settings, we demonstrate that Adam suffers from lack of momentum tuning while closed-loop YellowFin performs significantly better.

We conduct experiments on both convolutional neural networks and recurrent neural networks. For convolutional neural networks, we evaluate on image recognition using a 110-layer ResNet [24] on CIFAR10 [25] and a 164-layer ResNet on CIFAR100. For diversity of the models, we use regular and bottleneck building units respectively for CIFAR10 and CIFAR100. For recurrent neural networks, we evaluate with LSTMs in 3 tasks: character-level language modeling with the TinyShakespeare (TS) dataset [26], word-level language modeling with the Penn TreeBank (PTB) [27], as well as constituency parsing on the Wall Street Journal (WSJ) dataset [28]. We refer to Table 2 in Appendix E for detailed specification of model architectures.

5.1 Synchronous experiments

In our synchronous experiments, we tune momentum SGD and Adam on learning rate grids with momentum set to 0.9 for momentum SGD. We fix the parameters of Algorithm 1 in all experiments, i.e. YellowFin runs without any hand tuning. We provide full specifications, including the learning rate (grid) and the number of iterations we train on each model in Appendix F for reproducibility. For visualization purposes, we smooth all the training losses with an uniform window of width 1000. For Adam and momentum SGD on each model, we pick the configuration achieving the lowest smoothed loss in corresponding grids. To compare two algorithms, we record the lowest smoothed loss achieved by both. Then the speedup is reported as the ratio of iterations to achieve this loss. We use this setup to validate the following two claims.

Table 1: The speedup of YellowFin and tuned mom. SGD comparing to tuned Adam.

	Adam	mom. SGD	YellowFin
CIFAR10	1x	1.58x	2.41x
CIFAR100	1x	1.28x	2.82x
PTB	1x	0.91x	1.18x
TS	1x	1.94x	2.16x
WSJ	1x	1.36x	2.01x

Momentum SGD is competitive with adaptive methods Adaptive methods are justifiably popular as they reduce the need to tune hyperparameters, but they do not necessarily achieve faster convergence. In Figure 5 and Table 1, we present the comparison between tuned momentum SGD and tuned Adam on ResNets. We can observe that momentum SGD achieves 1.58x and 1.28x speedup on CIFAR10 and CIFAR100 ResNets respectively comparing to tuned Adam. Similarly in Figure 6 and Table 1, except from converging slightly slower than Adam for PTB LSTM, momentum SGD consistently produces measurably better training loss, as well as better test perplexity in language modeling and test F1 in parsing. For the parsing task on WSJ, we also compare with tuned Vanilla SGD, which is more commonly used in the NLP community. As seen from Figure 6

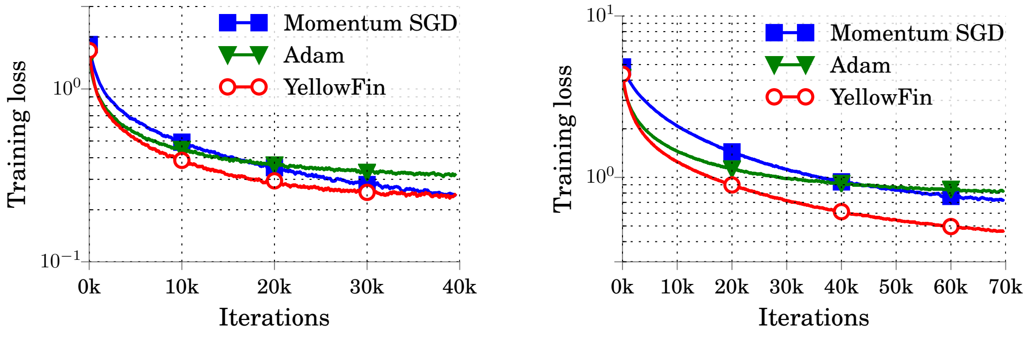

Figure 5: Training loss for ResNet on CIFAR10 (left) and CIFAR100 (right).

(right), *fixed momentum 0.9 can already speedup Vanilla SGD by 3.45x and achieve observably better test F1*. These observations show that momentum is critical to acceleration, and momentum SGD can be better than the state-of-the-art adaptive method in a variety of models.

YellowFin can outperform hand-tuned momentum SGD and hand-tuned Adam on ResNets and LSTMs In our experiments, YellowFin, without any hand-tuning, yields speedups from to 1.15x to 2.34x on ResNets and LSTMs in comparison to tuned momentum SGD. When comparing to tuned Adam in Table 1, YellowFin achieves speedups from 1.18x to 2.82x in training losses. *More importantly,* YellowFin *consistently shows better test metrics than tuned momentum SGD and Adam.* It demonstrates that YellowFin can outperform hand-tuned state-of-the-art adaptive and non-adaptive optimizers.

5.2 Asynchronous experiments

In this section we measure the performance of YellowFin in an asynchronous-parallel setting, where we focus on *statistical efficiency*: the number of iterations to reach a certain solution. To that end, we run M asynchronous workers on a single machine and force them to update the model in a round-robin fashion, i.e. the staled gradient is delayed for $(M-1)$ iterations. We demonstrate (1) Adam suffers a convergence speed penalty due to not tuning momentum in asynchronous settings; (2) closed-loop YellowFin (cf. Section 4) improves the convergence of YellowFin dramatically and this leads to (3) closed-loop YellowFin achieving much faster convergence than Adam.

State-of-the-art adaptive algorithms do not tune momentum We conduct experiments on PTB LSTM with 16 asynchronous workers using Adam. Fixing the learning rate to the value achieving the lowest smoothed loss in Section 5.1, we sweep the smoothing parameter β_1 [14] of the first order moment estimate in grid $\{-0.2, 0.0, 0.3, 0.5, 0.7, 0.9\}$. β_1 serves the same role as momentum in SGD and we call it the momentum in Adam. Figure 7 (left) shows tuning momentum for Adam under asynchrony gives measurably better training loss. This result emphasizes the importance of

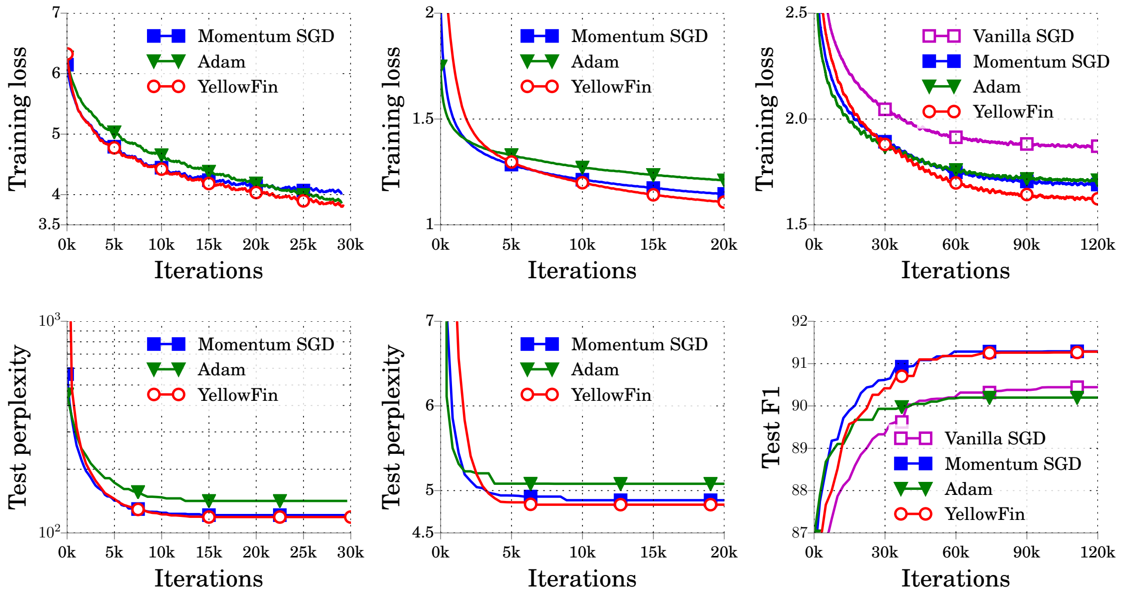

Figure 6: Training loss and test metrics on word-level language modeling with PTB (left), character-level language modeling with TS (middle) and constituency parsing on WSJ (right). Note the test metrics are monotonic as we report the best values up to each specific number of iterations.

momentum tuning in asynchronous settings and suggests that state-of-the-art adaptive methods pay a penalty for using prescribed momentum.

Closing the loop improves convergence under staleness We compare the performance of YellowFin in Algorithm 1 and closed-loop YellowFin in Algorithm 5 on the 164-layer bottleneck ResNet. We conduct experiments using 16 asynchronous workers. In Figure 7 (right), we observe closed-loop YellowFin achieves more than 2x speedup to reach loss 3.0 and 1.3x for the lowest loss from YellowFin. This demonstrates that closed-loop YellowFin accelerates by effectively reducing algorithmic momentum to compensate for asynchrony. We notice that closed-loop YellowFin achieves very similar loss as the synchronous baseline near the end. It suggests an almost 16x wall-clock time speedup by parallelizing asynchronously.

Closed-loop YellowFin outperforms Adam in asynchrony We run Adam with 16 workers on CIFAR100 ResNet using the learning rate achieving the lowest smoothed loss in Section 5.1. Shown in Figure 7 (right), Adam slows down dramatically due to asynchrony, while closed-loop YellowFin, with feedback gain $\gamma = 0.01$, demonstrates up to 2.7x speedup over Adam. It shows closed-loop YellowFin can significantly outperform the state-of-the-art in asynchronous settings.

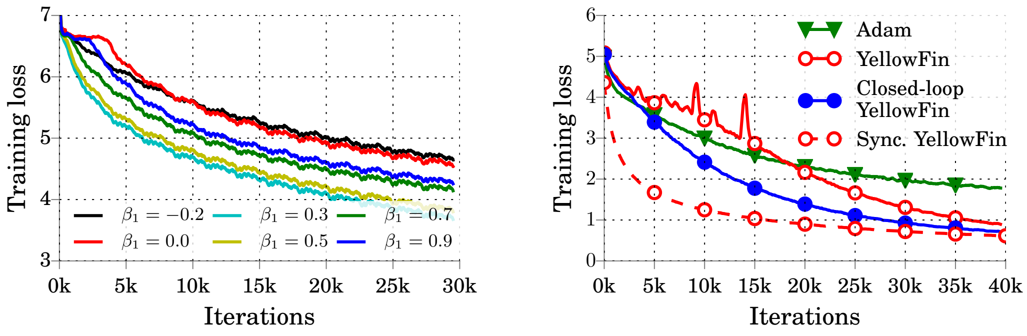

Figure 7: Hand-tuning Adam's momentum under asynchrony (left) on PTB LSTM. Asynchronous performance comparison with YellowFin in synchronous settings as a baseline (right) on CIFAR100.

6 Related work

Many techniques have been proposed on tuning hyperparameters for optimizers. Bergstra and Bengio [10] investigate random search for general tuning of hyperparameters. Bayesian approaches [11] model evaluation metrics as samples from a Gaussian process guiding optimal hyperparameter search. Another trend is the adaptive methods which require less manual tuning than SGD: Adagrad [12] is one of the first method with per-dimension learning rate, followed by RMSProp [13] and Adam [17] using different learning rate rules. Schaul et al. [29] use a noisy quadratic model similar to ours, however they tune per-variable learning rates and do not use momentum.

7 Discussion

We presented YellowFin, the first optimization method that automatically tunes momentum as well as the learning rate of momentum SGD. YellowFin outperforms the state-of-the-art in optimizing a large class of models both in synchronous and asynchronous settings. It estimates statistics purely from the gradients of a running system, and then tunes the hyperparameters of momentum SGD based on noisy, local quadratic approximations. As future work, we believe that more accurate curvature estimation methods, like the *bbprop* method [30] can further improve YellowFin. We also believe that our closed-loop momentum control mechanism in Section 4 could be applied to other adaptive methods and improve performances in asynchronous-parallel settings.

8 Acknowledgements

We thank Bryan He, Paroma Varma, Chris De Sa and Theodoros Rekatsinas for helpful discussions. We gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) SIMPLEX program under No. N66001-15-C-4043, the D3M program under No. FA8750-17-2-0095, the National Science Foundation (NSF) CAREER Award under No. IIS- 1353606, the Office of

Naval Research (ONR) under awards No. N000141210041 and No. N000141310129, a Sloan Research Fellowship, the Moore Foundation, an Okawa Research Grant, Toshiba, and Intel. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, NSF, ONR, or the U.S. government.

References

[1] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. *arXiv preprint arXiv:1705.08292*, 2017.

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

[3] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony begets momentum, with an application to deep learning. *arXiv preprint arXiv:1605.09774*, 2016.

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

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

[6] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[7] Genevieve B Orr and Klaus-Robert Müller. *Neural networks: tricks of the trade*. Springer, 2003.

[8] Yoshua Bengio et al. Deep learning of representations for unsupervised and transfer learning. *ICML Unsupervised and Transfer Learning*, 27:17–36, 2012.

[9] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[10] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[11] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[12] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[13] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2), 2012.

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

[15] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[16] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[17] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pages 571–582, 2014.

[18] Stefan Hadjis, Ce Zhang, Ioannis Mitliagkas, Dan Iter, and Christopher Ré. Omnivore: An optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*, 2016.

[19] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

[20] Laurent Lessard, Benjamin Recht, and Andrew Packard. Analysis and design of optimization algorithms via integral quadratic constraints. *SIAM Journal on Optimization*, 26(1):57–95, 2016.

[21] John Duchi. Fisher information., 2016. URL `https://web.stanford.edu/class/stats311/Lectures/lec-09.pdf`.

[22] Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*, 2013.

[23] Ce Zhang and Christopher Ré. Dimmwitted: A study of main-memory statistical analytics. *PVLDB*, 7(12):1283–1294, 2014. URL `http://www.vldb.org/pvldb/vol7/p1283-zhang.pdf`.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[25] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset, 2014.

[26] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[27] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

[28] Do Kook Choe and Eugene Charniak. Parsing as language modeling.

[29] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. *ICML (3)*, 28: 343–351, 2013.

[30] James Martens, Ilya Sutskever, and Kevin Swersky. Estimating the hessian by back-propagating curvature. *arXiv preprint arXiv:1206.6464*, 2012.

A Proof of Lemma 2

To prove Lemma 2, we first prove a more generalized version in Lemma 6. By restricting f to be a one dimensional quadratics function, the generalized curvature h_t itself is the only eigenvalue. We can prove Lemma 2 as a straight-forward corollary. Lemma 6 also implies, in the multiple dimensional correspondence of (4), the spectral radius $\rho(\boldsymbol{A}_t) = \sqrt{\mu}$ if the curvature on all eigenvector directions (eigenvalue) satisfies (5).

Lemma 6. *Let the gradients of a function f be described by*

$$\nabla f(\boldsymbol{x}_t) = \boldsymbol{H}(\boldsymbol{x}_t)(\boldsymbol{x}_t - \boldsymbol{x}^*), \tag{16}$$

with $\boldsymbol{H}(\boldsymbol{x}_t) \in \mathbb{R}^n \mapsto \mathbb{R}^{n\times n}$. Then the momentum update can be expressed as a linear operator:

$$\begin{pmatrix} \boldsymbol{y}_{t+1} \\ \boldsymbol{y}_t \end{pmatrix} = \begin{pmatrix} \boldsymbol{I} - \alpha \boldsymbol{H}(\boldsymbol{x}_t) + \mu \boldsymbol{I} & -\mu \boldsymbol{I} \\ \boldsymbol{I} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{y}_t \\ \boldsymbol{y}_{t-1} \end{pmatrix} = \boldsymbol{A}_t \begin{pmatrix} \boldsymbol{y}_t \\ \boldsymbol{y}_{t-1} \end{pmatrix}, \tag{17}$$

where $\boldsymbol{y}_t \triangleq \boldsymbol{x}_t - \boldsymbol{x}^*$. Now, assume that the following condition holds for all eigenvalues $\lambda(\boldsymbol{H}(\boldsymbol{x}_t))$ of $\boldsymbol{H}(\boldsymbol{x}_t)$:

$$\frac{(1-\sqrt{\mu})^2}{\alpha} \le \lambda(\boldsymbol{H}(\boldsymbol{x}_t)) \le \frac{(1+\sqrt{\mu})^2}{\alpha}. \tag{18}$$

then the spectral radius of $\boldsymbol{A}_t$ is controlled by momentum with $\rho(\boldsymbol{A}_t) = \sqrt{\mu}$.

Proof. Let λ_t be an eigenvalue of matrix $\boldsymbol{A}_t$, it gives $\det(\boldsymbol{A}_t - \lambda_t \boldsymbol{I}) = 0$. We define the blocks in $\boldsymbol{A}_t$ as $\boldsymbol{C} = \boldsymbol{I} - \alpha \boldsymbol{H}_t + \mu \boldsymbol{I} - \lambda_t \boldsymbol{I}$, $\boldsymbol{D} = -\mu \boldsymbol{I}$, $\boldsymbol{E} = \boldsymbol{I}$ and $\boldsymbol{F} = -\lambda_t \boldsymbol{I}$ which gives

$$\det(\boldsymbol{A}_t - \lambda_t \boldsymbol{I}) = \det \boldsymbol{F} \det\left(\boldsymbol{C} - \boldsymbol{D}\boldsymbol{F}^{-1}\boldsymbol{E}\right) = 0$$

assuming generally $\boldsymbol{F}$ is invertible. Note we use $\boldsymbol{H}_t \triangleq \boldsymbol{H}(\boldsymbol{x}_t)$ for simplicity in writing. The equation $\det\left(\boldsymbol{C} - \boldsymbol{D}\boldsymbol{F}^{-1}\boldsymbol{E}\right) = 0$ implies that

$$\det\left(\lambda_t^2 \boldsymbol{I} - \lambda_t \boldsymbol{M}_t + \mu \boldsymbol{I}\right) = 0 \tag{19}$$

with $\boldsymbol{M}_t = (\boldsymbol{I} - \alpha \boldsymbol{H}_t + \mu \boldsymbol{I})$. In other words, λ_t satisfied that $\lambda_t^2 - \lambda_t \lambda(\boldsymbol{M}_t) + \mu = 0$ with $\lambda(\boldsymbol{M}_t)$ being one eigenvalue of $\boldsymbol{M}_t$. I.e.

$$\lambda_t = \frac{\lambda(\boldsymbol{M}_t) \pm \sqrt{\lambda(\boldsymbol{M}_t)^2 - 4\mu}}{2} \tag{20}$$

On the other hand, (18) guarantees that $(1 - \alpha\lambda(\boldsymbol{H}_t) + \mu)^2 \le 4\mu$. We know both $\boldsymbol{H}_t$ and $\boldsymbol{I} - \alpha\boldsymbol{H}_t + \mu\boldsymbol{I}$ are symmetric. Thus for all eigenvalues $\lambda(\boldsymbol{M}_t)$ of $\boldsymbol{M}_t$, we have $\lambda(\boldsymbol{M}_t)^2 = (1 - \alpha\lambda(\boldsymbol{H}_t) + \mu)^2 \le 4\mu$ which guarantees $|\lambda_t| = \sqrt{\mu}$ for all λ_t. As the spectral radius is equal to the magnitude of the largest eigenvalue of $\boldsymbol{A}_t$, we have the spectral radius of $\boldsymbol{A}_t$ being $\sqrt{\mu}$.

$\square$

B Proof of Lemma 4

We first prove Lemma 7 and Lemma 8 as preparation for the proof of Lemma 4. After the proof for one dimensional case, we discuss the trivial generalization to multiple dimensional case.

Lemma 7. *Let the h be the curvature of a one dimensional quadratic function f and $\overline{x}_t = \mathbb{E}x_t$. We assume, without loss of generality, the optimum point of f is $x^\star = 0$. Then we have the following recurrence*

$$\begin{pmatrix} \overline{x}_{t+1} \\ \overline{x}_t \end{pmatrix} = \begin{pmatrix} 1-\alpha h+\mu & -\mu \\ 1 & 0 \end{pmatrix}^t \begin{pmatrix} x_1 \\ x_0 \end{pmatrix} \tag{21}$$

Proof. From the recurrence of momentum SGD, we have

$$\begin{aligned} \mathbb{E}x_{t+1} =& \mathbb{E}[x_t - \alpha \nabla f_{S_t}(x_t) + \mu(x_t - x_{t-1})] \\ =& \mathbb{E}_{x_t}[x_t - \alpha \mathbb{E}_{S_t} \nabla f_{S_t}(x_t) + \mu(x_t - x_{t-1})] \\ =& \mathbb{E}_{x_t}[x_t - \alpha h x_t + \mu(x_t - x_{t-1})] \\ =& (1-\alpha h+\mu)\overline{x}_t - \mu \overline{x}_{t-1} \end{aligned}$$

By putting the equation in to matrix form, (21) is a straight-forward result from unrolling the recurrence for t times. Note as we set $x_1 = x_0$ with no uncertainty in momentum SGD, we have $[\overline{x}_0, \overline{x}_1] = [x_0, x_1]$. □

Lemma 8. *Let $U_t = \mathbb{E}(x_t - \overline{x}_t)^2$ and $V_t = \mathbb{E}(x_t - \overline{x}_t)(x_{t-1} - \overline{x}_{t-1})$ with $\overline{x}_t$ being the expectation of x_t. For quadratic function $f(x)$ with curvature $h \in \mathbb{R}$, We have the following recurrence*

$$\begin{pmatrix} U_{t+1} \\ U_t \\ V_{t+1} \end{pmatrix} = (\boldsymbol{I} - \boldsymbol{B}^\top)(\boldsymbol{I} - \boldsymbol{B})^{-1} \begin{pmatrix} \alpha^2 C \\ 0 \\ 0 \end{pmatrix} \tag{22}$$

where

$$\boldsymbol{B} = \begin{pmatrix} (1-\alpha h+\mu)^2 & \mu^2 & -2\mu(1-\alpha h+\mu) \\ 1 & 0 & 0 \\ 1-\alpha h+\mu & 0 & -\mu \end{pmatrix} \tag{23}$$

and $C = \mathbb{E}(\nabla f_{S_t}(x_t) - \nabla f(x_t))^2$ is the variance of gradient on minibatch S_t.

Proof. We prove by first deriving the recurrence for U_t and V_t respectively and combining them in to a matrix form. For U_t, we have

$$\begin{aligned} U_{t+1} =& \mathbb{E}(x_{t+1} - \overline{x}_{t+1})^2 \\ =& \mathbb{E}(x_t - \alpha \nabla f_{S_t}(x_t) + \mu(x_t - x_{t-1}) - (1-\alpha h+\mu)\overline{x}_t + \mu \overline{x}_{t-1})^2 \\ =& \mathbb{E}(x_t - \alpha \nabla f(x_t) + \mu(x_t - x_{t-1}) - (1-\alpha h+\mu)\overline{x}_t + \mu \overline{x}_{t-1} + \alpha(\nabla f(x_t) - \nabla f_{S_t}(x_t)))^2 \\ =& \mathbb{E}((1-\alpha h+\mu)(x_t - \overline{x}_t) - \mu(x_{t-1} - \overline{x}_{t-1}))^2 + \alpha^2 \mathbb{E}(\nabla f(x_t) - \nabla f_{S_t}(x_t))^2 \\ =& (1-\alpha h+\mu)^2 \mathbb{E}(x_t - \overline{x}_t)^2 - 2\mu(1-\alpha h+\mu)\mathbb{E}(x_t - \overline{x}_t)(x_{t-1} - \overline{x}_{t-1}) \\ & + \mu^2 \mathbb{E}(x_{t-1} - \overline{x}_{t-1})^2 + \alpha^2 C \end{aligned} \tag{24}$$

where the cross terms cancels due to the fact $\mathbb{E}_{S_t}[\nabla f(x_t) - \nabla f_{S_t}(x_t)] = 0$ in the third equality.

For V_t, we can similarly derive

$$\begin{aligned} V_t =& \mathbb{E}(x_t - \overline{x}_t)(x_{t-1} - \overline{x}_{t-1}) \\ =& \mathbb{E}((1-\alpha h + \mu)(x_{t-1} - \overline{x}_{t-1}) - \mu(x_{t-2} - \overline{x}_{t-2}) + \alpha(\nabla f(x_t) - \nabla f_{S_t}(x_t)))(x_{t-1} - \overline{x}_{t-1}) \\ =& (1-\alpha h + \mu)\mathbb{E}(x_{t-1} - \overline{x}_{t-1})^2 - \mu \mathbb{E}(x_{t-1} - \overline{x}_{t-1})(x_{t-2} - \overline{x}_{t-2}) \end{aligned} \tag{25}$$

Again, the term involving $\nabla f(x_t) - \nabla f_{S_t}(x_t)$ cancels in the third equality as a results of $\mathbb{E}_{S_t}[\nabla f(x_t) - \nabla f_{S_t}(x_t)] = 0$. (24) and (25) can be jointly expressed in the following matrix form

$$\begin{pmatrix} U_{t+1} \\ U_t \\ V_{t+1} \end{pmatrix} = \boldsymbol{B} \begin{pmatrix} U_t \\ U_{t-1} \\ V_t \end{pmatrix} + \begin{pmatrix} \alpha^2 C \\ 0 \\ 0 \end{pmatrix} = \sum_{i=0}^{t-1} \boldsymbol{B}^i \begin{pmatrix} \alpha^2 C \\ 0 \\ 0 \end{pmatrix} + \boldsymbol{B}^t \begin{pmatrix} U_1 \\ U_0 \\ V_1 \end{pmatrix} = (\boldsymbol{I} - \boldsymbol{B}^t)(\boldsymbol{I} - \boldsymbol{B})^{-1} \begin{pmatrix} \alpha^2 C \\ 0 \\ 0 \end{pmatrix}. \tag{26}$$

Note the second term in the second equality is zero because x_0 and x_1 are deterministic. Thus $U_1 = U_0 = V_1 = 0$. □

According to Lemma 7 and 8, we have $\mathbb{E}(\overline{x}_t - x^*)^2 = (\boldsymbol{e}_1^\top \boldsymbol{A}^t [x_1, x_0]^\top)^2$ and $\mathbb{E}(x_t - \overline{x}_t)^2 = \alpha^2 C \boldsymbol{e}_1^\top (\boldsymbol{I} - \boldsymbol{B}^t)(\boldsymbol{I} - \boldsymbol{B})^{-1} \boldsymbol{e}_1$ where $\boldsymbol{e}_1 \in \mathbb{R}^n$ has all zero entries but the first dimension. Combining these two terms, we prove Lemma 4. Though the proof here is for one dimensional quadratics, it trivially generalizes to multiple dimensional quadratics. Specifically, we can decompose the quadratics along the eigenvector directions, and then apply Lemma 4 to each eigenvector direction using the corresponding curvature h (eigenvalue). By summing quantities in (10) for all eigenvector directions, we can achieve the multiple dimensional correspondence of (10).

C Proof of Lemma 5

Again we first present a proof of a multiple dimensional generalized version of Lemma 5. The proof of Lemma 5 is a one dimensional special case of Lemma 9. Lemma 9 also implies that for multiple dimension quadratics, the corresponding spectral radius $\rho(\boldsymbol{B}) = \mu$ if $\frac{(1-\sqrt{\mu})^2}{\alpha} \le h \le \frac{(1+\sqrt{\mu})^2}{\alpha}$ on all the eigenvector directions with h being the eigenvalue (curvature).

Lemma 9. *Let $\boldsymbol{H} \in \mathbb{R}^{n\times n}$ be a symmetric matrix and $\rho(\boldsymbol{B})$ be the spectral radius of matrix*

$$\boldsymbol{B} = \begin{pmatrix} (\boldsymbol{I} - \alpha \boldsymbol{H} + \mu \boldsymbol{I})^\top (\boldsymbol{I} - \alpha \boldsymbol{H} + \mu \boldsymbol{I}) & \mu^2 \boldsymbol{I} & -2\mu(\boldsymbol{I} - \alpha \boldsymbol{H} + \mu \boldsymbol{I}) \\ \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{I} - \alpha \boldsymbol{H} + \mu \boldsymbol{I} & \boldsymbol{0} & -\mu \boldsymbol{I} \end{pmatrix} \tag{27}$$

We have $\rho(\boldsymbol{B}) = \mu$ if all eigenvalues $\lambda(\boldsymbol{H})$ of $\boldsymbol{H}$ satisfies

$$\frac{(1-\sqrt{\mu})^2}{\alpha} \le \lambda(\boldsymbol{H}) \le \frac{(1+\sqrt{\mu})^2}{\alpha}. \tag{28}$$

Proof. Let λ be an eigenvalue of matrix $\boldsymbol{B}$, it gives $\det(\boldsymbol{B} - \lambda \boldsymbol{I}) = 0$ which can be alternatively expressed as

$$\det(\boldsymbol{B} - \lambda \boldsymbol{I}) = \det \boldsymbol{F} \det\left(\boldsymbol{C} - \boldsymbol{D}\boldsymbol{F}^{-1}\boldsymbol{E}\right) = 0 \tag{29}$$

assuming $\boldsymbol{F}$ is invertible, i.e. $\lambda + \mu \neq 0$, where the blocks in $\boldsymbol{B}$

$$\boldsymbol{C} = \begin{pmatrix} \boldsymbol{M}^\top \boldsymbol{M} - \lambda \boldsymbol{I} & \mu^2 \boldsymbol{I} \\ \boldsymbol{I} & -\lambda \boldsymbol{I} \end{pmatrix}, \boldsymbol{D} = \begin{pmatrix} -2\mu \boldsymbol{M} \\ \boldsymbol{0} \end{pmatrix}, \boldsymbol{E} = \begin{pmatrix} \boldsymbol{M} \\ \boldsymbol{0} \end{pmatrix}^\top, \boldsymbol{F} = -\mu \boldsymbol{I} - \lambda \boldsymbol{I}$$

with $\boldsymbol{M} = \boldsymbol{I} - \alpha \boldsymbol{H} + \mu \boldsymbol{I}$. (29) can be transformed using straight-forward algebra as

$$\det \begin{pmatrix} (\lambda - \mu)\boldsymbol{M}^\top \boldsymbol{M} - (\lambda + \mu)\lambda \boldsymbol{I} & (\lambda + \mu)\mu^2 \boldsymbol{I} \\ (\lambda + \mu)\boldsymbol{I} & -(\lambda + \mu)\lambda \boldsymbol{I} \end{pmatrix} \tag{30}$$

Using similar simplification technique as in (29), we can further simplify into

$$(\lambda - \mu) \det \Big((\lambda + \mu)^2 \boldsymbol{I} - \lambda \boldsymbol{M}^\top \boldsymbol{M} \Big) = 0 \tag{31}$$

if $\lambda \neq \mu$, as $(\lambda+\mu)^2 \boldsymbol{I} - \lambda \boldsymbol{M}^\top \boldsymbol{M}$ is diagonalizable, we have $(\lambda+\mu)^2 - \lambda\lambda(\boldsymbol{M})^2 = 0$ with $\lambda(\boldsymbol{M})$ being an eigenvalue of symmetric $\boldsymbol{M}$. The analytic solution to the equation can be explicitly expressed as

$$\lambda = \frac{\lambda(\boldsymbol{M})^2 - 2\mu \pm \sqrt{(\lambda(\boldsymbol{M})^2 - 2\mu)^2 - 4\mu^2}}{2}. \tag{32}$$

When the condition in (28) holds, we have $\lambda(M)^2 = (1 - \alpha\lambda(\boldsymbol{H}) + \mu)^2 \leq 4\mu$. One can verify that

$$\begin{aligned} (\lambda(\boldsymbol{M})^2 - 2\mu)^2 - 4\mu^2 = & \ (\lambda(\boldsymbol{M})^2 - 4\mu)\lambda(\boldsymbol{M})^2 \\ = & \ \big((1 - \alpha\rho(\boldsymbol{H}) + \mu)^2 - 4\mu\big)\, \lambda(\boldsymbol{M})^2 \\ \leq & \ 0 \end{aligned} \tag{33}$$

Thus the roots in (32) are conjugate with $|\lambda| = \mu$. In conclusion, the condition in (28) can guarantee all the eigenvalues of $\boldsymbol{B}$ has magnitude μ. Thus the spectral radius of $\boldsymbol{B}$ is controlled by μ. $\square$

D Analytical solution to (14)

The problem in (14) does not need iterative solver but has an analytical solution. Substituting only the second constraint, the objective becomes $p(x) = x^2 D^2 + (1-x)^4 / h_{\min}^2 C$ with $x = \sqrt{\mu} \in [0, 1)$. By setting the gradient of $p(x)$ to 0, we can get a cubic equation whose root $x = \sqrt{\mu_p}$ can be computed in closed form. As $p(x)$ is uni-modal in $[0, 1)$, the optimizer for (14) is exactly the maximum of μ_p and $(\sqrt{h_{\max}/h_{\min}} - 1)^2 / (\sqrt{h_{\max}/h_{\min}} + 1)^2$, the right hand-side of the first constraint in (14).

E Model specification

The model specification is shown in Table 2 for all the experiments in Section 5. CIRAR10 ResNet uses the regular ResNet units while CIFAR100 ResNet uses the bottleneck units. Only the convolutional layers are shown with filter size, filter number as well as the repeating count of the units. The layer counting for ResNets also includes batch normalization and Relu layers. The LSTM models are also diversified for different tasks with different vocabulary sizes, word embedding dimensions and number of layers.

Table 2: Specification of ResNet and LSTM model architectures.

network	# layers	Conv 0	Unit 1s	Unit 2s	Unit 3s
CIFAR10 ResNet	110	$\begin{bmatrix} 3\times 3, & 4 \end{bmatrix}$	$\begin{bmatrix} 3\times 3, & 4 \\ 3\times 3, & 4 \end{bmatrix} \times 6$	$\begin{bmatrix} 3\times 3, & 8 \\ 3\times 3, & 8 \end{bmatrix} \times 6$	$\begin{bmatrix} 3\times 3, & 16 \\ 3\times 3, & 16 \end{bmatrix} \times 6$
CIFAR100 ResNet	164	$\begin{bmatrix} 3\times 3, & 4 \end{bmatrix}$	$\begin{bmatrix} 1\times 1, & 16 \\ 3\times 3, & 16 \\ 1\times 1, & 64 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times 1, & 32 \\ 3\times 3, & 32 \\ 1\times 1, & 128 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times 1, & 64 \\ 3\times 3, & 64 \\ 1\times 1, & 256 \end{bmatrix} \times 6$

network	# layers	Word Embed.	Layer 1	Layer 2
TS LSTM	2	[65 vocab, 128 dim]	128 hidden units	128 hidden units
PTB LSTM	2	[10000 vocab, 200 dim]	200 hidden units	200 hidden units
WSJ LSTM	3	[6922 vocab, 500 dim]	500 hidden units	500 hidden units

F Specification for synchronous experiments

In Section 5.1, we demonstrate the synchronous experiments with extensive discussions. For the reproducibility, we provide here the specification of learning rate grids. The number of iterations as well as epochs, i.e. the number of passes over the full training sets, are also listed for completeness. For YellowFin, we uniformly set initial learning rate and momentum to $\alpha = 1.0$ and $\mu = 0.0$ for all the experiments in Section 5. For momentum SGD and Adam, we use the following configurations.

- CIFAR10 ResNet
  - 40k iterations (∼114 epochs)
  - Momentum SGD learning rates $\{0.001, 0.01(\text{best}), 0.1, 1.0\}$, momentum 0.9
  - Adam learning rates $\{0.0001, 0.001(\text{best}), 0.01, 0.1\}$
- CIFAR100 ResNet
  - 70k iterations (∼199 epochs)
  - Momentum SGD learning rates $\{0.001, 0.01(\text{best}), 0.1, 1.0\}$, momentum 0.9
  - Adam learning rates $\{0.0001, 0.001(\text{best}), 0.01, 0.1\}$
- PTB LSTM
  - 30k iterations (∼13 epochs)
  - Momentum SGD learning rates $\{0.01, 0.1, 1.0(\text{best}), 10.0\}$, momentum 0.9
  - Adam learning rates $\{0.0001, 0.001(\text{best}), 0.01, 0.1\}$
- TS LSTM
  - ∼21k iterations (50 epochs)
  - Momentum SGD learning rates $\{0.05, 0.1, 0.5, 1.0(\text{best}), 5.0\}$, momentum 0.9
  - Adam learning rates $\{0.0005, 0.001, 0.005, 0.01(\text{best}), 0.05\}$
  - Decrease learning rate by factor 0.97 every epoch for all optimizers, following the design by Karpathy et al. [26].

- WSJ LSTM
  - $\sim$120k iterations (50 epochs)
  - Vanilla SGD learning rates $\{0.05, 0.1, 0.5, 1.0(\text{best}), 5.0\}$
  - Momentum SGD learning rates $\{0.05, 0.1, 0.5(\text{best}), 1.0, 5.0\}$, momentum 0.9
  - Adam learning rates $\{0.0001, 0.0005, 0.001(\text{best}), 0.005, 0.01\}$
  - Decrease learning rate by factor 0.9 every epochs after 14 epochs for all optimizers, following the design by Choe and Charniak [28].