LipschitzLR: Using theoretically computed adaptive learning rates for fast convergence

Rahul Yedida[a], Snehanshu Saha[b,*]

[a]*Department of Computer Science, North Carolina State University, Raleigh, USA*
[b]*Department of Computer Science, PES University, Bengaluru, India*

Abstract

We present a novel theoretical framework for computing large, adaptive learning rates. Our framework makes minimal assumptions on the activations used and exploits the functional properties of the loss function. Specifically, we show that the inverse of the Lipschitz constant of the loss function is an ideal learning rate. We analytically compute formulas for the Lipschitz constant of several loss functions, and through extensive experimentation, demonstrate the strength of our approach using several architectures and datasets. In addition, we detail the computation of learning rates when other optimizers, namely, SGD with momentum, RMSprop, and Adam, are used. Compared to standard choices of learning rates, our approach converges faster, and yields better results.

Keywords: Lipschitz constant, adaptive learning, machine learning, deep learning

1. Introduction

Gradient descent[1] is a popular optimization algorithm for finding optima for functions, and is used to find optima in loss functions in machine learning tasks. In an iterative process, it seeks to update randomly initialized weights to minimize the training error. These updates are typically small values proportional to the gradient of the loss function. The constant of proportionality is called the learning rate, and is usually manually chosen.

*Corresponding author.
Email address: snehanshusaha@pes.edu (Snehanshu Saha)

In this paper, we propose a novel theoretical framework to compute large, adaptive learning rates for use in gradient-based optimization algorithms. We start with a presentation of the theoretical framework and the motivation behind it, and then derive the mathematical formulas to compute the learning rate on each epoch. We then extend our approach from stochastic gradient descent (SGD) to other optimization algorithms. Finally, we present extensive experimental results to support our claims.

Our experimental results show that compared to standard choices of learning rates, our approach converges quicker and achieves better results. During the experiments, we explore cases where adaptive learning rates outperform fixed learning rates. Our approach exploits functional properties of the loss function, and only makes two minimal assumptions on the loss function: it must be Lipschitz continuous[2] and (at least) once differentiable. Commonly used loss functions satisfy both these properties.

In summary, our contributions in this paper are as follows:

- We present a theoretical framework based on the Lipschitz constant of the loss function to compute an adaptive learning rate.
- We provide an intuitive motivation for using the inverse of the Lipschitz constant as the learning rate in gradient-based optimization algorithms, and derive formulas for the Lipschitz constant of several commonly used loss functions. We note that classical machine learning models, such as logistic regression, are simply special cases of deep learning models, and show the equivalence of the formulas derived.
- Through extensive experimentation, we demonstrate the strength of our results with both classical machine learning models and deep learning models.

The rest of the paper is organized as follows. Related work on optimization and deep learning is presented in section 2. This is followed by our theoretical framework based on the properties of several loss functions, in sections 3 and 4. Section 5 elaborates application of the framework in binary and multi-class classification, with a note on regularization. We extend the framework to algorithms that extend SGD, such as RMSprop, momentum, and Adam in section 6. Section 7 details data sets, experimental settings and results obtained. We conclude in section 9 preceded by a short note on practical considerations.

2. Related Work

2.1. Optimization algorithms

The gradient descent update rule is given by

$$\mathbf{w} := \mathbf{w} - \alpha \cdot \nabla_{\mathbf{w}} f$$

The generalization ability of stochastic gradient descent (SGD) and various methods of faster optimization have quickly gained interest in machine learning and deep learning communities.

Several directions have been taken to understand these phenomena. The interest in the stability of SGD is one such direction[3, 4]. Others have proven that gradient descent can find the global minima of the loss functions in over-parameterized deep neural networks[5, 6].

More practical approaches in this regard have involved novel changes to the optimization procedure itself. These include adding a "momentum" term to the update rule [7], and "adaptive gradient" methods such as RMSProp[8], and Adam[9]. These methods have seen widespread use in deep neural networks[10, 11, 12]. Other methods rely on an approximation of the Hessian. These include the Broyden-Fletcher-Goldfarb-Shanno (BFGS) [13, 14, 15, 16] and L-BFGS[17] algorithms. However, our proposed method does not require any modification of the standard gradient descent update rule, and only schedules the learning rate. Furthermore, for classical machine learning models, this learning rate is fixed and thus, our approach does not take any extra time. In addition, we only use the first gradient, thus requiring functions to be only once differentiable and L-Lipschitz.

2.2. Deep learning

Deep learning [18] is becoming more omnipresent for several tasks, including image recognition and classification [19, 20, 21, 22], face recognition [23], and object detection [24], even surpassing human-level performance[25]. At the same time, the trend is towards deeper neural networks [26, 25].

Despite their popularity, training neural networks is made difficult by several problems. These include vanishing and exploding gradients [27, 28] and overfitting. Various advances including different activation functions [29, 30], batch normalization [26], novel initialization schemes [25], and dropout [31] offer solutions to these problems.

However, a more fundamental problem is that of finding optimal values for various hyperparameters, of which the learning rate is arguably the most

important. It is well-known that learning rates that are too small are slow to converge, while learning rates that are too large cause divergence [32]. Recent works agree that rather than a fixed learning rate value, a non-monotonic learning rate scheduling system offers faster convergence [33, 34]. It has also been argued that the traditional wisdom that large learning rates should not be used may be flawed, and can lead to "super-convergence" and have regularizing effects [35]. Our experimental results agree with this statement; however, rather than use cyclical learning rates based on intuition, we propose a novel method to compute an adaptive learning rate backed by theoretical foundations.

Recently, there has been a lot of work on finding novel ways to adaptively change the learning rate. These have both theoretical [33] and intuitive, empirical [35, 34] backing. These works rely on non-monotonic scheduling of the learning rate. [34] argues for cyclical learning rates. Our proposed method also yields a non-monotonic learning rate, but does not follow any predefined shape.

3. Theoretical Framework

3.1. Introduction and Motivation

For a function, the Lipschitz constant is the least positive constant L such that

$$\|f(\mathbf{w}_1) - f(\mathbf{w}_2)\| \leq L \|\mathbf{w}_1 - \mathbf{w}_2\|$$

for all $\mathbf{w}_1$, $\mathbf{w}_2$ in the domain of f. From the mean-value theorem for scalar fields, for any $\mathbf{w}_1, \mathbf{w}_2$, there exists $\mathbf{v}$ such that

$$\begin{aligned} \|f(\mathbf{w}_1) - f(\mathbf{w}_2)\| &= \|\nabla_{\mathbf{w}} f(\mathbf{v})\| \|\mathbf{w}_1 - \mathbf{w}_2\| \\ &\leq \sup_{\mathbf{v}} \|\nabla_{\mathbf{w}} f(\mathbf{v})\| \|\mathbf{w}_1 - \mathbf{w}_2\| \end{aligned}$$

Thus, $\sup_{\mathbf{v}} \|\nabla_{\mathbf{w}} f(\mathbf{v})\|$ is such an L. Since L is the least such constant,

$$L \leq \sup_{\mathbf{v}} \|\nabla_{\mathbf{w}} f(\mathbf{v})\|$$

In this paper, we use $\max \|\nabla_{\mathbf{w}} f\|$ to derive the Lipschitz constants. Our approach makes the minimal assumption that the functions are Lipschitz

continuous and differentiable up to first order only [1]. Because the gradient of these loss functions is used in gradient descent, these conditions are guaranteed to be satisfied.

By setting $\alpha = \frac{1}{L}$, we have $\Delta \mathbf{w} \leq 1$, constraining the change in the weights. We stress here that we are not computing the Lipschitz constants of the *gradients* of the loss functions, but of the losses themselves. Therefore, our approach merely assumes the loss is L-Lipschitz, and not β-smooth. We argue that the boundedness of the effective weight changes makes it optimal to set the learning rate to the reciprocal of the Lipschitz constant. This claim, while rather bold, is supported by our experimental results.

3.2. Notation

We use the following notation:

- $(x^{(i)}, y^{(i)})$ refers to one training example. The superscript with parentheses indicates the ith training example. X refers to the input matrix.
- Where not specified, it should be assumed that m indicates the number of training examples.
- For deep neural networks, whenever unclear, we use a superscript with square brackets to indicate the layer number. For example, $W^{[l]}$ indicates the weight matrix at the lth layer. We use L to represent the total number of layers, being careful not to cause ambiguity with the Lipschitz constant.
- We use the letter w or W to refer to weights, while b refers to a bias term. Capital letters indicate matrices; lowercase letters indicate scalars, and are usually accompanied by subscripts–in such a case, we will adequately describe what the subscripts indicate.
- We use the letter a to denote an activation; thus, $a^{[l]}$ represents the activations at the lth layer.

[1] Note this is a weaker condition than assuming the gradient of the function being Lipschitz continuous. We exploit merely the boundedness of the gradient.

3.3. Deriving the Lipschitz constant for neural networks

For a neural network that uses the sigmoid, ReLU, or softmax activations, it is easily shown that the gradients get smaller towards the earlier layers in backpropagation. Because of this, the gradients at the last layer are the maximum among all the gradients computed during backpropagation. If $w_{ij}^{[l]}$ is the weight from node i to node j at layer l, and if L is the number of layers, then

$$\max_{h,k} \frac{\partial E}{\partial w_{hk}^{[L]}} \geq \frac{\partial E}{\partial w_{ij}^{[l]}} \forall\ l, i, j \tag{1}$$

Essentially, (1) says that the maximum gradient of the error with respect to the weights in the last layer is greater than the gradient of the error with respect to any weight in the network. In other words, finding the maximum gradient at the last layer gives us a supremum of the Lipschitz constants of the error, where the gradient is taken with respect to the weights at any layer. We call this supremum as a Lipschitz constant of the loss function for brevity.

We now analytically arrive at a theoretical Lipschitz constant for different types of problems. The inverse of these values can be used as a learning rate in gradient descent. Specifically, since the Lipschitz constant that we derive is an upper bound on the gradients, we effectively limit the size of the parameter updates, without necessitating an overly guarded learning rate. In any layer, we have the computations

$$z^{[l]} = W^{[l]T} a^{[l-1]} + b^{[l]} \tag{2}$$

$$a^{[l]} = g(z^{[l]}) \tag{3}$$

$$a^{[0]} = X \tag{4}$$

Thus, the gradient with respect to any weight in the last layer is computed via the chain rule as follows.

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}^{[L]}} &= \frac{\partial E}{\partial a_j^{[L]}} \cdot \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \cdot \frac{\partial z_j^{[L]}}{\partial w_{ij}^{[L]}} \\ &= \frac{\partial E}{\partial a_j^{[L]}} \cdot \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \cdot a_j^{[L-1]} \end{aligned} \tag{5}$$

This gives us

$$\max_{i,j}\left\|\frac{\partial E}{\partial w_{ij}^{[L]}}\right\| = \max_{j}\left\|\frac{\partial E}{\partial a_{j}^{[L]}}\right\| \cdot \max_{j}\left\|\frac{\partial a_{j}^{[L]}}{\partial z_{j}^{[L]}}\right\| \cdot \max_{j}\left\|a_{j}^{[L-1]}\right\| \tag{6}$$

The third part cannot be analytically computed; we denote it as K_z. We now look at various types of problems and compute these components. Note that we use the terms "cost function" and "loss function" interchangeably.

4. Least-squares cost function

For the least squares cost function, we will separately compute the Lipschitz constant for a linear regression model and for neural networks where the output is continuous. We will then prove the equivalence of the two results, deriving the former as a special case of the latter.

4.1. Linear regression

We have,

$$g(\mathbf{w}) = \frac{1}{2m}\sum_{i=1}^{m}\left(\mathbf{x}^{(i)}\mathbf{w} - y^{(i)}\right)^2$$

Thus,

$$\begin{aligned}
g(\mathbf{w}) - g(\mathbf{v}) &= \frac{1}{2m}\sum_{i=1}^{m}\left(\mathbf{x}^{(i)}\mathbf{w} - y^{(i)}\right)^2 - \left(\mathbf{x}^{(i)}\mathbf{v} - y^{(i)}\right)^2 \\
&= \frac{1}{2m}\sum_{i=1}^{m}\left(\mathbf{x}^{(i)}(\mathbf{w}+\mathbf{v}) - 2y^{(i)}\right)\left(\mathbf{x}^{(i)}(\mathbf{w}-\mathbf{v})\right) \\
&= \frac{1}{2m}\sum_{i=1}^{m}\left((\mathbf{w}+\mathbf{v})^T\mathbf{x}^{(i)T} - 2y^{(i)}\right)\left(\mathbf{x}^{(i)}(\mathbf{w}-\mathbf{v})\right) \\
&= \frac{1}{2m}\sum_{i=1}^{m}\left((\mathbf{w}+\mathbf{v})^T\mathbf{x}^{(i)T}\mathbf{x}^{(i)} - 2y^{(i)}\mathbf{x}^{(i)}\right)(\mathbf{w}-\mathbf{v})
\end{aligned}$$

The penultimate step is obtained by observing that $(\mathbf{w}+\mathbf{v})^T\mathbf{x}^{(i)T}$ is a real number, whose transpose is itself.

At this point, we take the norm on both sides, and then assume that $\mathbf{w}$ and $\mathbf{v}$ are bounded such that $\|\mathbf{w}\|, \|\mathbf{v}\| \leq K$. Taking norm on both sides,

$$\boxed{\frac{\|g(\mathbf{w}) - g(\mathbf{v})\|}{\|\mathbf{w} - \mathbf{v}\|} \leq \frac{K}{m}\left\|\mathbf{X}^T\mathbf{X}\right\| - \frac{1}{m}\left\|\mathbf{y}^T\mathbf{X}\right\|}$$

We are forced to use separate norms because the matrix subtraction $2K\mathbf{X}^T\mathbf{X} - 2\mathbf{y}^T\mathbf{X}$ cannot be performed. The RHS here is the Lipschitz constant. Note that the Lipschitz constant changes if the cost function is considered with a factor other than $\frac{1}{2m}$.

4.2. Regression with neural networks

Let the loss be given by

$$E(\mathbf{a}^{[L]}) = \frac{1}{2m}\left(\mathbf{a}^{[L]} - \mathbf{y}\right)^2 \tag{7}$$

where the vectors contain the values for each training example. Then we have,

$$\begin{aligned} E(\mathbf{b}^{[L]}) - E(\mathbf{a}^{[L]}) &= \frac{1}{2m}\left(\left(\mathbf{b}^{[L]} - \mathbf{y}\right)^2 - \left(\mathbf{a}^{[L]} - \mathbf{y}\right)^2\right) \\ &= \frac{1}{2m}\left(\mathbf{b}^{[L]} + \mathbf{a}^{[L]} - 2\mathbf{y}\right)\left(\mathbf{b}^{[L]} - \mathbf{a}^{[L]}\right) \end{aligned}$$

This gives us,

$$\begin{aligned} \frac{\|E(\mathbf{b}^{[L]}) - E(\mathbf{a}^{[L]})\|}{\|\mathbf{b}^{[L]} - \mathbf{a}^{[L]}\|} &= \frac{1}{2m}\|\mathbf{b}^{[L]} + \mathbf{a}^{[L]} - 2\mathbf{y}\| \\ &\leq \frac{1}{m}\left(K_a - \|\mathbf{y}\|\right) \end{aligned} \tag{8}$$

where K_a is the upper bound of $\|\mathbf{a}\|$ and $\|\mathbf{b}\|$. A reasonable choice of norm is the 2-norm.

Looking back at (6), the second term on the right side of the equation is the derivative of the activation with respect to its parameter. Notice that if the activation is sigmoid or softmax, then it is necessarily less than 1; if it is ReLu, it is either 0 or 1. Therefore, to find the maximum, we assume that the network is comprised solely of ReLu activations, and the maximum of this is 1.

From (6), we have

$$\boxed{\max_{i,j}\left\|\frac{\partial E}{\partial w_{ij}^{[L]}}\right\| = \frac{1}{m}\left(K_a - \|\mathbf{y}\|\right)K_z} \tag{9}$$

4.3. Equivalence of the constants

The equivalence of the above two formulas is easy to see by understanding the terms of (9). We had defined in (6),

$$K_z = \max_j\left\|a_j^{[L-1]}\right\| \tag{10}$$

Because a linear regression model can be thought of as a neural network with no hidden layers and a linear activation, and from (4), we have,

$$\mathbf{a}^{[L-1]} = \mathbf{a}^0 = \mathbf{X}$$

and therefore

$$K_z = \max_j\left\|a_j^{[L-1]}\right\| = \|\mathbf{X}\| \tag{11}$$

Next, observe that K_a is the upper bound of the final layer activations. For a linear regression model, we have the "activations" as the outputs: $\hat{\mathbf{y}} = \mathbf{W}^T\mathbf{X}$. Using the assumption that $\|\mathbf{W}\|$ has an upper bound K, we obtain

$$K_a = \max\left\|\mathbf{a}^{[L]}\right\| = \max\left\|\mathbf{W}^T\mathbf{X}\right\| = \max\|\mathbf{W}\|\cdot\|\mathbf{X}\| = K\|\mathbf{X}\| \tag{12}$$

Substituting (11) and (12) in (9), we obtain

$$\begin{aligned}\max_{i,j}\left\|\frac{\partial E}{\partial w_{ij}^{[L]}}\right\| &= \frac{1}{m}\left(K_a - \|\mathbf{y}\|\right)K_z \\ &= \frac{1}{m}\left(K\|\mathbf{X}\| - \|\mathbf{y}\|\right)\|\mathbf{X}\| \\ &= \frac{K}{m}\left\|\mathbf{X}^T\mathbf{X}\right\| - \frac{1}{m}\left\|\mathbf{y}^T\mathbf{X}\right\|\end{aligned}$$

□

This argument can also be used for the other loss functions that we discuss below; therefore, we will not prove equivalence of the Lipschitz constants for classical machine learning models (logistic regression and softmax regression) and neural networks. However, we will show experiments on both separately.

5. Classification

5.1. Binary classification

For binary classification, we use the binary cross-entropy loss function. Assuming only one output node,

$$E(\mathbf{z}^{[L]}) = -\frac{1}{m}\left(\mathbf{y}\log g(\mathbf{z}^{[L]}) + (1-\mathbf{y})\log(1-g(\mathbf{z}^{[L]}))\right) \tag{13}$$

where $g(z)$ is the sigmoid function. We use a slightly different version of (6) here:

$$\max_{i,j}\left|\frac{\partial E}{\partial w_{ij}^{[L]}}\right| = \max_{j}\left|\frac{\partial E}{\partial z_{j}^{[L]}}\right| \cdot K_z \tag{14}$$

Then, we have

$$\begin{aligned}
\frac{\partial E}{\partial \mathbf{z}^{[L]}} &= -\frac{1}{m}\left(\frac{\mathbf{y}}{g(\mathbf{z}^{[L]})}g(\mathbf{z}^{[L]})(1-g(\mathbf{z}^{[L]})) - \frac{1-\mathbf{y}}{1-g(\mathbf{z}^{[L]})}g(\mathbf{z}^{[L]})(1-g(\mathbf{z}^{[L]}))\right) \\
&= -\frac{1}{m}\left(\mathbf{y}(1-g(\mathbf{z}^{[L]})) - (1-\mathbf{y})g(\mathbf{z}^{[L]})\right) \\
&= -\frac{1}{m}\left(\mathbf{y} - \mathbf{y}g(\mathbf{z}^{[L]}) - g(\mathbf{z}^{[L]}) + \mathbf{y}g(\mathbf{z}^{[L]})\right) \\
&= -\frac{1}{m}\left(\mathbf{y} - g(\mathbf{z}^{[L]})\right)
\end{aligned} \tag{15}$$

It is easy to show, using the second derivative, that this attains a maxima at $\mathbf{z}^{[L]} = 0$:

$$\frac{\partial^2 E}{\partial \mathbf{w}_{ij}^{[L]2}} = \frac{1}{m}g(\mathbf{z}^{[L]})(1-g(\mathbf{z}^{[L]}))a_j^{[L-1]} \tag{16}$$

Setting (16) to 0 yields $a_j^{[L-1]} = 0\ \forall j$, and thus $z^{[L]} = W_{ij}^{[L]}a_j^{[L-1]} = 0$. This implies $g(\mathbf{z}^{[L]}) = \frac{1}{2}$. Now whether $\mathbf{y}$ is 0 or 1, substituting this back in (15), we get

$$\max_{j}\left\|\frac{\partial E}{\partial z_{j}^{[L]}}\right\| = \frac{1}{2m} \tag{17}$$

Using (17) in (14),

$$\boxed{\max_{i,j}\left\|\frac{\partial E}{\partial w_{ij}^{[L]}}\right\| = \frac{K_z}{2m}} \tag{18}$$

We simply mention here that for logistic regression, the Lipschitz constant is

$$\boxed{L = \frac{1}{2m}\|\mathbf{X}\|}$$

5.2. Multi-class classification

While conventionally, multi-class classification is done using one-hot encoded outputs, that is not convenient to work with mathematically. An identical form of this is to assume the output follows a Multinomial distribution, and then updating the loss function accordingly. This is because the effect of the typical loss function used is to only consider the "hot" vector; we achieve the same effect using the Iverson notation, which is equivalent to the Kronecker delta. With this framework, the loss function is

$$E(\mathbf{a}^{[L]}) = -\frac{1}{m}\sum_{j=1}^{k}[\mathbf{y} = j]\log \mathbf{a}^{[L]} \tag{19}$$

Then the first part of (6) is trivial to compute:

$$\frac{\partial E}{\partial \mathbf{a}^{[L]}} = -\frac{1}{m}\sum_{j=1}^{m}\frac{[\mathbf{y} = j]}{\mathbf{a}^{[L]}} \tag{20}$$

The second part is computed as follows.

$$\begin{aligned}
\frac{\partial a_j^{[L]}}{\partial z_p^{[L]}} &= \frac{\partial}{\partial z_p^{[L]}}\left(\frac{e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}}\right) \\
&= \frac{[p = j]e^{z_j^{[L]}}\sum_{l=1}^{k} e^{z_l^{[L]}} - e^{z_j^{[L]}} \cdot e^{z_p^{[L]}}}{\left(\sum_{l=1}^{k} e^{z_l^{[L]}}\right)^2} \\
&= \frac{[p = j]e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} - \frac{e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} \cdot \frac{e^{z_p^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} \\
&= \left([p = j]a_j^{[L]} - a_j^{[L]}a_p^{[L]}\right) \\
&= a_j^{[L]}([p = j] - a_p^{[L]})
\end{aligned} \tag{21}$$

Combining (20) and (21) in (5) gives

$$\frac{\partial E}{\partial W_p^{[L]}} = \frac{1}{m}\left(a_p^{[L]} - [\mathbf{y} = p]\right)K_z \tag{22}$$

It is easy to show that the limiting case of this is when all softmax values are equal and each $y^{(i)} = p$; using this and $a_p^{[L]} = \frac{1}{k}$ in (22) and combining with (6) gives us our desired result:

$$\boxed{\max_j \left\| \frac{\partial E}{\partial W_j^{[L]}} \right\| = \frac{k-1}{km} K_z} \tag{23}$$

For a softmax regression model, we have

$$\boxed{L = \frac{k-1}{km} \|\mathbf{X}\|}$$

5.3. Regularization

This framework is extensible to the case where the loss function includes a regularization term.

In particular, if an L_2 regularization term, $\frac{\lambda}{2} \|\mathbf{w}\|_2^2$ is added, it is trivial to show that the Lipschitz constant increases by λK, where K is the upper bound for $\|\mathbf{w}\|$. More generally, if a Tikhonov regularization term, $\|\mathbf{\Gamma w}\|_2^2$ term is added, then the increase in the Lipschitz constant can be computed as below.

$$\begin{aligned} L(\mathbf{w}_1) - L(\mathbf{w}_2) &= (\mathbf{\Gamma w}_1)^T (\mathbf{\Gamma w}_1) - (\mathbf{\Gamma w}_2)^T (\mathbf{\Gamma w}_2) \\ &= \mathbf{w}_1^T \mathbf{\Gamma}^2 \mathbf{w}_1 - \mathbf{w}_2^T \mathbf{\Gamma}^2 \mathbf{w}_2 \\ &= 2\mathbf{w}_2^T \mathbf{\Gamma}^2 (\mathbf{w}_1 - \mathbf{w}_2) + (\mathbf{w}_1 - \mathbf{w}_2)^T \Gamma^2 (\mathbf{w}_1 - \mathbf{w}_2) \\ \frac{\|L(\mathbf{w}_1) - L(\mathbf{w}_2)\|}{\|\mathbf{w}_1 - \mathbf{w}_2\|} &\leq 2 \|\mathbf{w}_2\| \left\|\mathbf{\Gamma}^2\right\| + \|\mathbf{w}_1 - \mathbf{w}_2\| \left\|\mathbf{\Gamma}^2\right\| \end{aligned}$$

If $\mathbf{w}_1, \mathbf{w}_2$ are bounded by K,

$$\boxed{L = 2K \left\|\mathbf{\Gamma}^2\right\|}$$

This additional term may be added to the Lipschitz constants derived above when gradient descent is performed on a loss function including a Tikhonov regularization term. Clearly, for an L_2-regularizer, since $\mathbf{\Gamma} = \frac{\lambda}{2}\mathbf{I}$, we have $L = \lambda K$.

6. Going Beyond SGD

The framework presented so far easily extends to algorithms that extend SGD, such as RMSprop, momentum, and Adam. In this section, we show algorithms for some major optimization algorithms popularly used.

RMSprop, gradient descent with momentum, and Adam are based on exponentially weighted averages of the gradients. The trick then is to compute the Lipschitz constant as an exponentially weighted average of the norms of the gradients. This makes sense, since it provides a supremum of the "velocity" or "accumulator" terms in momentum and RMSprop respectively.

6.1. Gradient Descent with Momentum

SGD with momentum uses an exponentially weighted average of the gradient as a velocity term. The gradient is replaced by the velocity in the weight update rule.

Algorithm 1: AdaMo

1 $K \leftarrow 0$; $V_{\nabla L} \leftarrow 0$;
2 **for** *each iteration* **do**
3 Compute $\nabla_W L$ for all layers;
4 $V_{\nabla L} \leftarrow \beta V_{\nabla L} + (1-\beta)\nabla_W L$;
5 `// Compute the exponentially weighted average of LC`
6 $K \leftarrow \beta K + (1-\beta)\max\|\nabla_W L\|$;
7 `// Weight update`
8 $W \leftarrow W - \frac{1}{K} V_{\nabla L}$;
9 **end**

Algorithm 1 shows the *adaptive* version of gradient descent with momentum. The only changes are on lines 6 and 8. The exponentially weighted average of the Lipschitz constant ensures that the learning rate for that iteration is optimal. The weight update is changed to reflect our new learning rate. We use the symbol W to consistently refer to the weights as well as the biases; while "parameters" may be a more apt term, we use W to stay consistent with literature.

Notice that only line 6 is our job; deep learning frameworks will typically take care of the rest; we simply need to compute K and use a learning rate scheduler that uses the inverse of this value.

6.2. RMSprop

RMSprop uses an exponentially weighted average of the square of the gradients. The square is performed element-wise, and thus preserves dimensions. The update rule in RMSprop replaces the gradient with the ratio of the current gradient and the exponentially moving average. A small value ϵ is added to the denominator for numerical stability.

Algorithm 2 shows the modified version of RMSprop. We simply maintain an exponentially weighted average of the Lipschitz constant as before; the learning rate is also replaced by the inverse of the update term, with the exponentially weighted average of the square of the gradient replaced with our computed exponentially weighted average.

Algorithm 2: Adaptive RMSprop

$K \leftarrow 0$; $S_{\nabla L} \leftarrow 0$;
for *each iteration* **do**
  Compute $\nabla_W L$ on mini-batch;
  $S_{\nabla L} \leftarrow \beta S_{\nabla L} + (1-\beta)(\nabla_W L)^2$;
  `// Compute the exponentially weighted average of LC`
  $K \leftarrow \beta K + (1-\beta)\max\|(\nabla_W L)^2\|$;
  `// Weight update`
  $W \leftarrow W - \frac{\sqrt{K}+\epsilon}{\max\|\nabla_W L\|} \cdot \frac{\nabla_W L}{\sqrt{S_{\nabla L}}+\epsilon}$;
end

6.3. Adam

Adam combines the above two algorithms. We thus need to maintain two exponentially weighted average terms. The algorithm, shown in Algorithm 3, is quite straightforward.

Algorithm 3: Auto-Adam

$K_1 \leftarrow 0$; $K_2 \leftarrow 0$; $S_{\nabla L} \leftarrow 0$; $V_{\nabla L} = 0$;
for *each iteration* **do**
  Compute $\nabla_W L$ on mini-batch;
  $V_{\nabla L} \leftarrow \beta_1 V_{\nabla L} + (1-\beta_1)\nabla_W L$;
  $S_{\nabla L} \leftarrow \beta_2 S_{\nabla L} + (1-\beta_2)(\nabla_W L)^2$;
  `// Compute the exponentially weighted averages of LC`
  $K_1 \leftarrow \beta_1 K_1 + (1-\beta_1)\max\|\nabla_W L\|$;
  $K_2 \leftarrow \beta_2 K_2 + (1-\beta_2)\max\|(\nabla_W L)^2\|$;
  `// Weight update`
  $W \leftarrow W - \frac{\sqrt{K_2}+\epsilon}{K_1} \cdot \frac{V_{\nabla L}}{\sqrt{S_{\nabla L}}+\epsilon}$;
end

In our experiments, we use the defaults of $\beta_1 = 0.9, \beta_2 = 0.999$.

In practice, it is difficult to get a good estimate of $\max\|(\nabla_W L)^2\|$. For this reason, we tried two different estimates:

- $\|(\max \nabla_W L)^2\| = \left\| \left(\frac{k-1}{km} K_z + \lambda \|w\|\right)^2 \right\|$ – This set the learning rate high (around 4 on CIFAR-10 with DenseNet), and the model quickly diverged.
- $(\max\|\nabla_W L\|)^2 = \frac{(k-1)^2}{k^2 m^2} \max K_z^2 + \lambda^2 (\max\|w\|)^2 + \frac{2\lambda(k-1)}{km} K_z (\max\|w\|)$ – This turned out to be an overestimation, and while the same model above did not diverge, it oscillated around a local minimum. We fixed this by removing the middle term. This worked quite well empirically.

6.4. A note on bias correction

Some implementations of the above algorithms perform bias correction as well. This involves computing the exponentially weighted average, and then dividing by $1-\beta^t$, where t is the epoch number. In this case, the above algorithms may be adjusted by also dividing the Lipschitz constants by the same constant.

7. Experiments

In this section, we show through extensive experimentation that our approach converges faster, and performs better than with a standard choice of learning rate.

7.1. Faster convergence

For checking the rate of convergence, we use classical machine learning models. In each experiment, we randomly initialize weights, and use the same initial weight vector for gradient descent with both the learning rates. In all experiments, we scale each feature to sum to 1 before running gradient descent. This scaled data is used to compute the Lipschitz constants, and consequently, the learning rates. Normalizing the data is particularly important because the Lipschitz constant may get arbitrarily large, thus making the learning rate too small.

The regression experiments use a multiple linear regression model, the binary classification experiments use an ordinary logistic regression model, and the multi-class classification experiments use a softmax regression model with one-hot encoded target labels. For MNIST, however, we found it quicker to train a neural network with only an input and output layer (no hidden layers were used), a stochastic gradient descent optimizer, and softmax activations.

We compare the rate of convergence by setting a threshold, T_L for the value of the loss function. When the value of the cost function goes below this threshold, we stop the gradient descent procedure. A reasonable threshold value is chosen for each dataset separately. We then compare $E_{0.1}$ and $E_{1/L}$, where E_α represents the number of epochs taken for the loss to go below T_L. For the Cover Type data, we considered only the first two out of seven classes. This resulted in 495,141 rows. We also considered only ten features to speed up computation time.

For the least-squares cost function, an estimate of K is required. A good estimate of K would be obtained by running gradient descent with some fixed learning rate and then taking the norm of the final weight vectors. However, because this requires actually running the algorithm for which we want to find a parameter first, we need to estimate this value instead. In our experiments, we obtain a close approximation to the value obtained above through the formula below. For the experiments in this subsection, we use this formula to compute K.

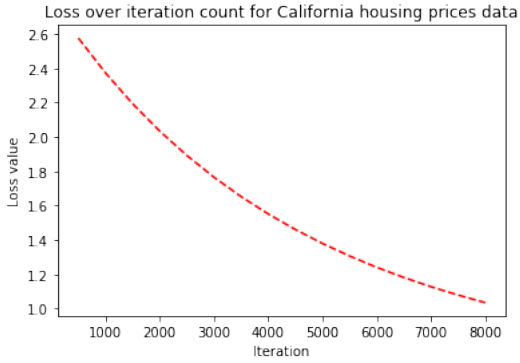

Figure 1: Loss function over iterations for California housing prices dataset

$$a = \frac{1}{m} \sum_{j=1}^{n} \sum_{i=1}^{m} x_j^{(i)}$$

$$b = \frac{1}{n} \sum_{j=1}^{n} \max_i x_j^{(i)}$$

$$K = \frac{a+b}{2}$$

In the above formulas, the notation $x_j^{(i)}$ refers to the jth column of the ith training example. Note that a is the sum of the means of each column, and b is the mean of the maximum of each column.

Table 1 shows the results of our experiments on some datasets. Clearly, our choice of α outperforms a random guess in all the datasets. Our proposed method yields a learning rate that adapts to each dataset to converge significantly faster than with a guess. In some datasets, our choice of learning rate gives over a 100x improvement in training time.

While the high learning rates may raise concerns of oscillations rather than convergence, we have checked for this in our experiments. To do this, we continued running gradient descent, monitoring the value of the loss function every 500 iterations. Figure 1 shows this plot, demonstrating that the high learning rates indeed lead to convergence.

[2]We restricted the data to the first 100K rows only.

Table 1: Comparison of speed of convergence with $\alpha = 0.1$ and $\alpha = \frac{1}{L}$

Dataset	T_L	$1/L$	$E_{0.1}$	$E_{1/L}$
Boston housing prices	200	9.316	46,041	555
California housing prices	2.8051	5163.5	24,582	2
Energy efficiency [36]	100	12.78	489,592	3,833
Online news popularity [37]	73,355,000	1.462	10,985	753
Breast cancer	0.69	4280.23	37,008	2
Covertype[2]	0.69314	17.48M	216,412	2
Iris	0.2	1.902	413	49
Digits	0.2	0.634	337	2

7.2. Better performance

We tested the performance of our approach with both classical machine learning models and deep neural networks. In this section, we discuss the results of both. To compare, we ran the models with different learning rates for a fixed number of epochs, N_E, and compared the accuracy scores $A_{0.1}$ and $A_{1/L}$, where A_α is the accuracy score after N_E epochs with the learning rates 0.1 and $1/L$ respectively.

Table 2 shows the results of these experiments. Figure 2 shows a comparative plot of the training and validation accuracy scores for both learning rates on the MNIST dataset. In both the plots, the red line is for $\alpha = 0.1$, while the green line is for $\alpha = \frac{1}{L}$. Although our choice of learning rate starts off worse, it quickly (¡ 100 iterations) outperforms a learning rate of 0.1. Further, the validation accuracy has a *decreasing* tendency for $\alpha = 0.1$, while it is more stable for $\alpha = 1/L$.

7.3. Deep neural networks

We compared the performance of our approach with deep neural networks on standard datasets as well. While our results are not state of the art, our

[3]The inverse Lipschitz constant is different here because the number of rows was not restricted to 100K. Also, the inverse Lipschitz constant here is not a typo. The learning rate was indeed set to 189.35 million.

Table 2: Comparison of performance with $\alpha = 0.1$ and $\alpha = \frac{1}{L}$

Dataset	N_E	$1/L$	$A_{0.1}$	$A_{1/L}$
Iris	200	1.936	93.33%	97.78%
Digits	200	0.635	91.3%	94.63%
MNIST	200	10.24	92.7%	92.8%
Cover Type[3]	1000	189.35M	43.05%	57.21%
Breast cancer	1000	4280.22	43.23%	90.5%

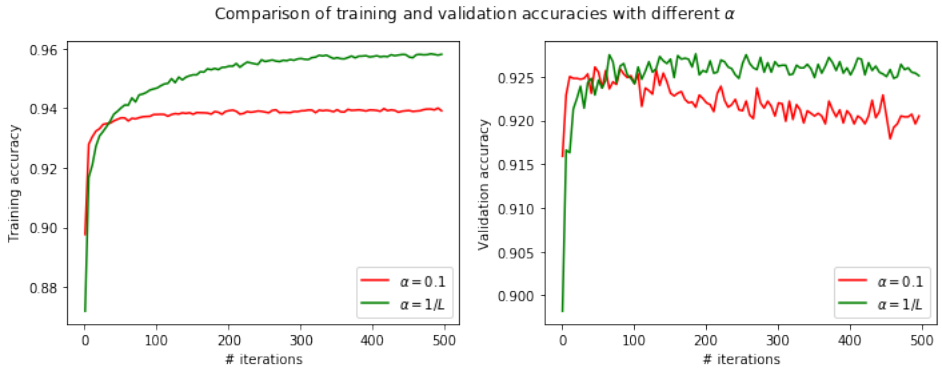

Figure 2: Comparison of training and validation accuracy scores for different α on MNIST

focus was to empirically show that optimization algorithms can be run with higher learning rates than typically understood. On CIFAR, we only use flipping and translation augmentation schemes as in [38]. In all experiments the raw image values were divided by 255 after removing the means across each channel. We also provide baseline experiments performed with a fixed learning rate for a fair comparison, using the same data augmentation scheme.

Table 3: Summary of all experiments: abbreviations used - LR: Learning Rate; WD: weight decay; VA: validation accuracy

Dataset	Architecture	Algorithm	LR Policy	WD	VA.
MNIST	Custom	SGD	Adaptive	None	99.5%
MNIST	Custom	Momentum	Adaptive	None	**99.57%**
MNIST	Custom	Adam	Adaptive	None	99.43%
CIFAR-10	ResNet20	SGD	Baseline	10^{-3}	60.33%
CIFAR-10	ResNet20	SGD	Fixed	10^{-3}	87.02%
CIFAR-10	ResNet20	SGD	Adaptive	10^{-3}	89.37%
CIFAR-10	ResNet20	Momentum	Baseline	10^{-3}	58.29%
CIFAR-10	ResNet20	Momentum	Adaptive	10^{-2}	84.71%
CIFAR-10	ResNet20	Momentum	Adaptive	10^{-3}	89.27%
CIFAR-10	ResNet20	RMSprop	Baseline	10^{-3}	84.92%
CIFAR-10	ResNet20	RMSprop	Adaptive	10^{-3}	86.66%
CIFAR-10	ResNet20	Adam	Baseline	10^{-3}	84.67%
CIFAR-10	ResNet20	Adam	Fixed	10^{-4}	70.57%
CIFAR-10	DenseNet	SGD	Baseline	10^{-4}	84.84%
CIFAR-10	DenseNet	SGD	Adaptive	10^{-4}	91.34%
CIFAR-10	DenseNet	Momentum	Baseline	10^{-4}	85.50%
CIFAR-10	DenseNet	Momentum	Adaptive	10^{-4}	**92.36%**
CIFAR-10	DenseNet	RMSprop	Baseline	10^{-4}	91.36%
CIFAR-10	DenseNet	RMSprop	Adaptive	10^{-4}	90.14%
CIFAR-10	DenseNet	Adam	Baseline	10^{-4}	91.38%
CIFAR-10	DenseNet	Adam	Adaptive	10^{-4}	88.23%
CIFAR-100	ResNet56	SGD	Adaptive	10^{-3}	54.29%

CIFAR-100	ResNet164	SGD	Baseline	10^{-4}	26.96%
CIFAR-100	ResNet164	SGD	Adaptive	10^{-4}	**75.99%**
CIFAR-100	ResNet164	Momentum	Baseline	10^{-4}	27.51%
CIFAR-100	ResNet164	Momentum	Adaptive	10^{-4}	75.39%
CIFAR-100	ResNet164	RMSprop	Baseline	10^{-4}	70.68%
CIFAR-100	ResNet164	RMSprop	Adaptive	10^{-4}	70.78%
CIFAR-100	ResNet164	Adam	Baseline	10^{-4}	71.96%
CIFAR-100	DenseNet	SGD	Baseline	10^{-4}	50.53%
CIFAR-100	DenseNet	SGD	Adaptive	10^{-4}	68.18%
CIFAR-100	DenseNet	Momentum	Baseline	10^{-4}	52.28%
CIFAR-100	DenseNet	Momentum	Adaptive	10^{-4}	69.18%
CIFAR-100	DenseNet	RMSprop	Baseline	10^{-4}	65.41%
CIFAR-100	DenseNet	RMSprop	Adaptive	10^{-4}	67.30%
CIFAR-100	DenseNet	Adam	Baseline	10^{-4}	66.05%
CIFAR-100	DenseNet	Adam	Adaptive	10^{-4}	40.14%[4]

A summary of our experiments is given in Table 3. DenseNet refers to a DenseNet[39] architecture with $L = 40$ and $k = 12$.

7.3.1. MNIST

On MNIST, the architecture we used is shown in Table 4. All activations except the last layer are ReLU; the last layer uses softmax activations. The model has 730K parameters.

Our preprocessing involved random shifts (up to 10%), zoom (to 10%), and rotations (to 15°). We used a batch size of 256, and ran the model for 20 epochs. The experiment on MNIST used only an adaptive learning rate, where the Lipschitz constant, and therefore, the learning rate was recomputed every epoch. Note that this works even though the penultimate layer is a Dropout layer. No regularization was used during training. With these settings, we achieved a training accuracy of 98.57% and validation accuracy 99.5%.

[4]This was obtained after 67 epochs. After that, the performance deteriorated, and after 170 epochs, we stopped running the model. We also ran the model on the same architecture, but restricting the number of filters to 12, which yielded 59.08% validation accuracy.

Table 4: CNN used for MNIST

Layer	Filters	Padding
3 x 3 Conv	32	Valid
3 x 3 Conv	32	Valid
2 x 2 MaxPool	–	–
Dropout (0.2)	–	–
3 x 3 Conv	64	Same
3 x 3 Conv	64	Same
2 x 2 MaxPool	–	–
Dropout (0.25)	–	–
3 x 3 Conv	128	Same
Dropout (0.25)	–	–
Flatten	–	–
Dense (128)	–	–
BatchNorm	–	–
Dropout (0.25)	–	–
Dense (10)	–	–

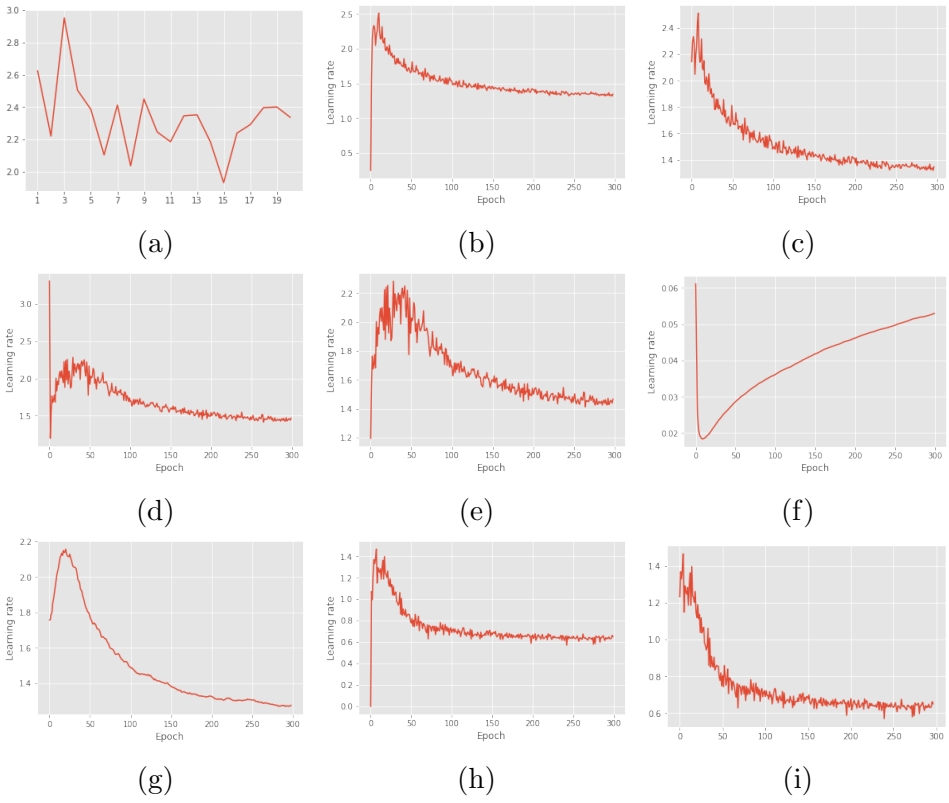

Figure 3: Plots of adaptive learning rate over time with various architectures and datasets. Where not specified, an SGD optimizer may be assumed. (a) Custom architecture on MNIST (b) ResNet20 on CIFAR-10 (c) ResNet20 on CIFAR-10 from epoch 2 (d) DenseNet on CIFAR-10 (e) DenseNet on CIFAR-10 from epoch 2 (f) DenseNet on CIFAR-10 with Adam optimizer (g) DenseNet on CIFAR-10 using AdaMo (h) ResNet164 on CIFAR-100 (i) ResNet164 on CIFAR-100 from epoch 3

Finally, Figure 3a shows the computed learning rate over epochs. Note that unlike the computed adaptive learning rates for CIFAR-10 (Figures 3b and 3c) and CIFAR-100 (Figures 3h and 3i), the learning rate for MNIST starts at a much higher value. While the learning rate here seems much more random, it must be noted that this was run for only 20 epochs, and hence any variation is exaggerated in comparison to the other models, run for 300 epochs.

The results of our Adam optimizer is also shown in Table 3. The optimizer achieved its peak validation accuracy after only 8 epochs.

We also used a custom implementation of SGD with momentum (see Appendix Appendix A for details), and computed an adaptive learning rate using our AdaMo algorithm. Surprisingly, this outperformed both our adaptive SGD and Auto-Adam algorithms. However, the algorithm consistently chose a large (around 32) learning rate for the first epoch before computing more reasonable learning rates–since this hindered performance, we modified our AdaMo algorithm so that on the first epoch, the algorithm sets K to 0.1 and uses this value as the learning rate. We discuss this issue further in Section 7.4.

7.4. CIFAR-10

For the CIFAR-10 experiments, we used a ResNet20 v1[38]. A residual network is a deep neural network that is made of "residual blocks". A residual block is a special case of a highway networks [40] that do not contain any gates in their skip connections. ResNet v2 also uses "bottleneck" blocks, which consist of a 1x1 layer for reducing dimension, a 3x3 layer, and a 1x1 layer for restoring dimension [41]. More details can be found in the original ResNet papers [38, 41].

We ran two sets of experiments on CIFAR-10 using SGD. First, we empirically computed K_z by running one epoch and finding the activations of the penultimate layer. We ran our model for 300 epochs using the same fixed learning rate. We used a batch size of 128, and a weight decay of 10^{-3}. Our computed values of K_z, $\max\|w\|$, and learning rate were 206.695, 43.257, and 0.668 respectively. It should be noted that while computing the Lipschitz constant, m in the denominator must be set to the batch size, not the total number of training examples. In our case, we set it to 128.

Figure 4 shows the plots of accuracy score and loss over time. As noted in [42], a horizontal validation loss indicates little overfitting. We achieved a

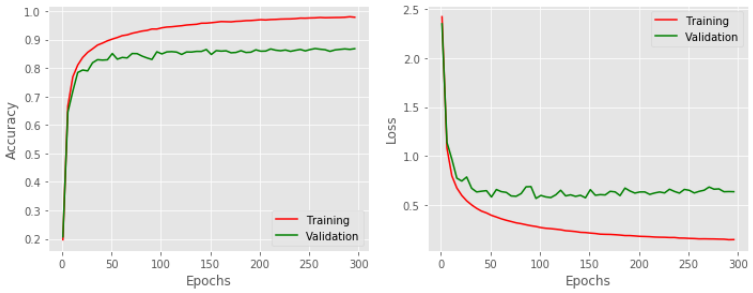

Figure 4: Plot of accuracy score and loss over epochs on CIFAR-10

training accuracy of 97.61% and a validation accuracy of 87.02% with these settings.

Second, we used the same hyperparameters as above, but recomputed K_z, $\max\|w\|$, and the learning rate every epoch. We obtained a training accuracy of 99.47% and validation accuracy of 89.37%. Clearly, this method is superior to a fixed learning rate policy.

Figure 3b and 3c show the learning rate over time. The adaptive scheme automatically chooses a decreasing learning rate, as suggested by literature on the subject. On the first epoch, however, the model chooses a very small learning rate of 8×10^{-3}, owing to the random initialization.

Observe that while it does follow the conventional wisdom of choosing a higher learning rate initially to explore the weight space faster and then slowing down as it approaches the global minimum, it ends up choosing a significantly larger learning rate than traditionally used. Clearly, there is no need to decay learning rate by a multiplicative factor. Our model with adaptive learning rate outperforms our model with a fixed learning rate in only 65 epochs. Further, the generalization error is lower with the adaptive learning rate scheme using the same weight decay value. This seems to confirm the notion in [35] that large learning rates have a regularization effect.

Figures 3d and 3e show the learning rate over time on CIFAR-10 using a DenseNet architecture and SGD. Evidently, the algorithm automatically adjusts the learning rate as needed.

Interestingly, in all our experiments, ResNets consistently performed poorly when run with our auto-Adam algorithm. Despite using fixed and adaptive learning rates, and several weight decay values, we could not optimize ResNets using auto-Adam. DenseNets and our custom architecture

on MNIST, however, had no such issues. Our best results with auto-Adam on ResNet20 and CIFAR-10 were when we continued using the learning rate of the first epoch (around 0.05) for all 300 epochs.

Figure 3f shows a possible explanation. Note that over time, our auto-Adam algorithm causes the learning rate to slowly increase. We postulate that this may be the reason for ResNet's poor performance using our auto-Adam algorithm. However, using SGD, we are able to achieve competitive results for all architectures. We discuss this issue further in Section 8.

ResNets did work well with our AdaMo algorithm, though, performing nearly as well as with SGD. As with MNIST, we had to set the initial learning rate to a fixed value with AdaMo. We find that a reasonable choice of this is between 0.1 and 1 (both inclusive). We find that for higher values of weight decay, lower values of x perform better, but we do not perform a more thorough investigation in this paper. In our experiments, we choose x by simply trying 0.1, 0.5, and 1.0, running the model for five epochs, and choosing the one that performs the best. In Table 3, for the first experiment using ResNet20 and momentum, we used $x = 0.1$; for the second, we used $x = 1$.

AdaMo also worked well with DenseNets on CIFAR-10. We used $x = 0.5$ for this model. This model crossed 90% validation accuracy before 100 epochs, maintaining a learning rate higher than 1, and was the best among all our models trained on CIFAR-10. This shows the strength of our algorithm. Figure 3g shows the learning rate over epochs for this model.

7.4.1. CIFAR-100

For the CIFAR-100 experiments, we used a ResNet164 v2 [41]. Our experiments on CIFAR-100 only used an adaptive learning rate scheme.

We largely used the same parameters as before. Data augmentation involved only flipping and translation. We ran our model for 300 epochs, with a batch size of 128. As in [41], we used a weight decay of 10^{-4}. We achieved a training accuracy of 99.68% and validation accuracy of 75.99% with these settings.

For the ResNet164 model trained using AdaMo, we found $x = 0.5$ to be the best among the three that we tried. Note that it performs competitively compared to SGD. For DenseNet, we used $x = 1$.

Figures 3h and 3i show the learning rate over epochs. As with CIFAR-10, the first two epochs start off with a very small (10^{-8}) learning rate, but the model quickly adjusts to changing weights.

7.4.2. Baseline Experiments

For our baseline experiments, we used the same weight decay value as our other experiments; the only difference was that we simply used a fixed value of the default learning rate for that experiment. For SGD and SGD with momentum, this meant a learning rate of 0.01. For Adam and RMSprop, the learning rate was 0.001. In SGD with momentum and RMSprop, $\beta = 0.9$ was used. For Adam, $\beta_1 = 0.9$ and $\beta_2 = 0.999$ were used.

8. Practical Considerations

Although our approach is theoretically sound, there are a few practical issues that need to be considered. In this section, we discuss these issues, and possible remedies.

The first issue is that our approach takes longer per epoch than with choosing a standard learning rate. Our code was based on the Keras deep learning library, which to the best of our knowledge, does not include a mechanism to get outputs of intermediate layers directly. Other libraries like PyTorch, however, do provide this functionality through "hooks". This eliminates the need to perform a partial forward propagation simply to obtain the penultimate layer activations, and saves computation time. We find that computing $\max\|w\|$ takes very little time, so it is not important to optimize its computation.

Another issue that causes practical issues is random initialization. Due to the random initialization of weights, it is difficult to compute the correct learning rate for the first epoch, because there is no data from a previous epoch to use. We discussed the effects of this already with respect to our AdaMo algorithm, and we believe this is the reason for the poor performance of auto-Adam in all our experiments. Fortunately, if this is the case, it can be spotted within the first two epochs–if large values of the intermediate computations: $\max\|w\|$, K_z, etc. are observed, then it may be required to set the initial LR to a suitable value. We discussed this for the AdaMo algorithm. In practice, we find that for RMSprop, this rarely occurs; but when it does, the large intermediate values are shown in the very first epoch. We find that a small value like 10^{-3} works well as the initial LR. In our experiments, we only had to do this for ResNet on CIFAR-100.

9. Discussion and Conclusion

In this paper, we derived a theoretical framework for computing an adaptive learning rate; on deriving the formulas for various common loss functions, it was revealed that this is also "adaptive" with respect to the data. We explored the effectiveness of this approach on several public datasets, with commonly used architectures and various types of layers.

Clearly, our approach works "out of the box" with various regularization methods including L_2, dropout, and batch normalization; thus, it does not interfere with regularization methods, and automatically chooses an optimal learning rate in stochastic gradient descent. On the contrary, we contend that our computed larger learning rates do indeed, as pointed out in [35], have a regularizing effect; for this reason, our experiments used small values of weight decay. Indeed, increasing the weight decay significantly hampered performance. This shows that "large" learning rates may not be harmful as once thought; rather, a large value may be used if carefully computed, along with a guarded value of L_2 weight decay. We also demonstrated the efficacy of our approach with other optimization algorithms, namely, SGD with momentum, RMSprop, and Adam.

Our auto-Adam algorithm performs surprisingly poorly. We postulate that like AdaMo, our auto-Adam algorithm will perform better when initialized more thoughtfully. To test this hypothesis, we re-ran the experiment with ResNet20 on CIFAR-10, using the same weight decay. We fixed the value of K_1 to 1, and found the best value of K_2 in the same manner as for AdaMo, but this time, checking 10^{-3}, 10^{-4}, 10^{-5}, and 10^{-6}. We found that the lower this value, the better our results, and we chose $K_2 = 10^{-6}$. While at this stage we can only conjecture that this combination of K_1 and K_2 will work in all cases, we leave a more thorough investigation as future work. Using this configuration, we achieved 83.64% validation accuracy.

A second avenue of future work involves obtaining a tighter bound on the Lipschitz constant and thus computing a more accurate learning rate. Another possible direction is to investigate possible relationships between the weight decay and the initial learning rate in the AdaMo algorithm.

Acknowledgements

Funding: This work was supported by the Science and Engineering Research Board (SERB)-Department of Science and Technology (DST), Government of India (project reference number SERB-EMR/ 2016/005687). The

funding source was not involved in the study design, writing of the report, or in the decision to submit this article for publication.

Appendix A. Implementation Details

All our code was written using the Keras deep learning library. The architecture we used for MNIST was taken from a Kaggle Python notebook by Aditya Soni[5]. For ResNets, we used the code from the Examples section of the Keras documentation[6]. The DenseNet implementation we used was from a GitHub repository by Somshubra Majumdar[7]. Finally, our implementation of SGD with momentum is a modified version of the Adam implementation in Keras[8].

References

[1] A. Cauchy, Méthode générale pour la résolution des systemes déquations simultanées, Comp. Rend. Sci. Paris 25 (1847) 536–538.

[2] S. Saha, Differential Equations: A structured Approach, Cognella, 2011.

[3] I. Kuzborskij, C. H. Lampert, Data-dependent stability of stochastic gradient descent, arXiv preprint arXiv:1703.01678 (2017).

[4] M. Hardt, B. Recht, Y. Singer, Train faster, generalize better: Stability of stochastic gradient descent, arXiv preprint arXiv:1509.01240 (2015).

[5] D. Zou, Y. Cao, D. Zhou, Q. Gu, Stochastic gradient descent optimizes over-parameterized deep relu networks, arXiv preprint arXiv:1811.08888 (2018).

[6] S. S. Du, X. Zhai, B. Poczos, A. Singh, Gradient descent provably optimizes over-parameterized neural networks, arXiv preprint arXiv:1810.02054 (2018).

[5] https://www.kaggle.com/adityaecdrid/mnist-with-keras-for-beginners-99457
[6] https://keras.io/examples/cifar10_resnet/
[7] https://github.com/titu1994/DenseNet
[8] https://github.com/keras-team/keras/blob/master/keras/optimizers.py#L436

[7] I. Sutskever, J. Martens, G. Dahl, G. Hinton, On the importance of initialization and momentum in deep learning, in: International conference on machine learning, pp. 1139–1147.

[8] T. Tieleman, G. Hinton, Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, COURSERA: Neural networks for machine learning 4 (2012) 26–31.

[9] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).

[10] A. Radford, L. Metz, S. Chintala, Unsupervised representation learning with deep convolutional generative adversarial networks, arXiv preprint arXiv:1511.06434 (2015).

[11] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, Y. Bengio, Show, attend and tell: Neural image caption generation with visual attention, in: International conference on machine learning, pp. 2048–2057.

[12] P. Bahar, T. Alkhouli, J.-T. Peter, C. J.-S. Brix, H. Ney, Empirical investigation of optimization algorithms in neural machine translation, The Prague Bulletin of Mathematical Linguistics 108 (2017) 13–25.

[13] C. G. Broyden, The convergence of a class of double-rank minimization algorithms: 2. the new algorithm, IMA journal of applied mathematics 6 (1970) 222–231.

[14] R. Fletcher, A new approach to variable metric algorithms, The computer journal 13 (1970) 317–322.

[15] D. Goldfarb, A family of variable-metric methods derived by variational means, Mathematics of computation 24 (1970) 23–26.

[16] D. F. Shanno, Conditioning of quasi-newton methods for function minimization, Mathematics of computation 24 (1970) 647–656.

[17] D. C. Liu, J. Nocedal, On the limited memory bfgs method for large scale optimization, Mathematical programming 45 (1989) 503–528.

[18] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT press, 2016.

[19] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556 (2014).

[20] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1–9.

[21] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, Y. LeCun, Overfeat: Integrated recognition, localization and detection using convolutional networks, arXiv preprint arXiv:1312.6229 (2013).

[22] M. D. Zeiler, R. Fergus, Visualizing and understanding convolutional networks, in: European conference on computer vision, Springer, pp. 818–833.

[23] Y. Taigman, M. Yang, M. Ranzato, L. Wolf, Deepface: Closing the gap to human-level performance in face verification, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1701–1708.

[24] R. Girshick, J. Donahue, T. Darrell, J. Malik, Rich feature hierarchies for accurate object detection and semantic segmentation, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 580–587.

[25] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, in: Proceedings of the IEEE international conference on computer vision, pp. 1026–1034.

[26] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, arXiv preprint arXiv:1502.03167 (2015).

[27] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Proceedings of the thirteenth international conference on artificial intelligence and statistics, pp. 249–256.

[28] Y. Bengio, P. Simard, P. Frasconi, et al., Learning long-term dependencies with gradient descent is difficult, IEEE transactions on neural networks 5 (1994) 157–166.

[29] G. Klambauer, T. Unterthiner, A. Mayr, S. Hochreiter, Self-normalizing neural networks, in: Advances in neural information processing systems, pp. 971–980.

[30] V. Nair, G. E. Hinton, Rectified linear units improve restricted boltzmann machines, in: Proceedings of the 27th international conference on machine learning (ICML-10), pp. 807–814.

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

[32] Y. Bengio, Neural networks: Tricks of the trade, Practical Recommendations for Gradient-Based Training of Deep Architectures, 2nd edn. Springer, Berlin, Heidelberg (2012) 437–478.

[33] S. Seong, Y. Lee, Y. Kee, D. Han, J. Kim, Towards flatter loss surface via nonmonotonic learning rate scheduling, in: UAI2018 Conference on Uncertainty in Artificial Intelligence, Association for Uncertainty in Artificial Intelligence (AUAI).

[34] L. N. Smith, Cyclical learning rates for training neural networks, in: 2017 IEEE Winter Conference on Applications of Computer Vision (WACV), IEEE, pp. 464–472.

[35] L. N. Smith, N. Topin, Super-convergence: Very fast training of neural networks using large learning rates, arXiv preprint arXiv:1708.07120 (2017).

[36] A. Tsanas, A. Xifara, Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools, Energy and Buildings 49 (2012) 560–567.

[37] K. Fernandes, P. Vinagre, P. Cortez, A proactive intelligent decision support system for predicting the popularity of online news, in: Portuguese Conference on Artificial Intelligence, Springer, pp. 535–546.

[38] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778.

[39] G. Huang, Z. Liu, L. Van Der Maaten, K. Q. Weinberger, Densely connected convolutional networks, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 4700–4708.

[40] R. K. Srivastava, K. Greff, J. Schmidhuber, Highway networks, arXiv preprint arXiv:1505.00387 (2015).

[41] K. He, X. Zhang, S. Ren, J. Sun, Identity mappings in deep residual networks, in: European conference on computer vision, Springer, pp. 630–645.

[42] L. N. Smith, A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay, arXiv preprint arXiv:1803.09820 (2018).