A novel adaptive learning rate scheduler for deep neural networks

Rahul Yedida and Snehanshu Saha

Abstract

Optimizing deep neural networks is largely thought to be an empirical process, requiring manual tuning of several hyper-parameters, such as learning rate, weight decay, and dropout rate. Arguably, the learning rate is the most important of these to tune, and this has gained more attention in recent works. In this paper, we propose a novel method to compute the learning rate for training deep neural networks with stochastic gradient descent. We first derive a theoretical framework to compute learning rates dynamically based on the Lipschitz constant of the loss function. We then extend this framework to other commonly used optimization algorithms, such as gradient descent with momentum and Adam. We run an extensive set of experiments that demonstrate the efficacy of our approach on popular architectures and datasets, and show that commonly used learning rates are an order of magnitude smaller than the ideal value.

1 Introduction

Deep learning [8] is becoming more omnipresent for several tasks, including image recognition [23, 30], face recognition [31], and object detection [6]. At the same time, the trend is towards deeper neural networks [13, 9].

Deep convolutional neural networks [16, 17] are a variant that introduce convolutional and pooling layers, and have seen incredible success in image classification [22, 34], even surpassing human-level performance [9]. Very deep convolutional neural networks have even crossed 1000 layers [11].

Despite their popularity, training neural networks is made difficult by several problems. These include vanishing and exploding gradients [7, 3] and overfitting. Various advances including different activation functions [15, 18], batch normalization [13], novel initialization schemes [9], and dropout [27] 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 [2]. Recent works agree that rather than a fixed learning rate value, a non-monotonic learning rate scheduling system offers faster convergence [21, 24]. 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 [26]. 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.

To the best of our knowledge, this is the first work to suggest an adaptive learning rate scheduler with a theoretical background and show experimental verification of its claim on standard datasets and network architectures. Thus, our contributions are as follows. First, we propose a novel theoretical framework for computing an optimal learning rate in stochastic gradient descent in deep neural networks, based on the Lipschitz constant of the loss function. We show that for certain choices of activation functions, only the activations in the last two layers are required to compute the learning rate. Second, we compute the ideal learning rate for several commonly used loss functions, and use these formulas to experimentally demonstrate the efficacy of our approach. Finally, we extend the above theoretical framework to derive adaptive versions of other common optimization algorithms, namely, gradient descent with momentum, RMSprop, and Adam. We also show experimental results using these algorithms.

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[20] and (at least) once differentiable. Commonly used loss functions satisfy both these properties.

The code, trained models, program outputs, and training history are available in our GitHub repository[1].

The rest of the paper is organized as follows. Section 2 discusses some related work. Section 3 introduces our novel theoretical framework. Sections 4 to 6 derive the learning rates for some common loss functions. Section 7 discusses how L_2 regularization fits into our proposed approach. Section 8 extends our framework to other optimization algorithms. Section 9 then shows experimental results demonstrating the benefits of our approach. Finally, Section 10 discusses some practical considerations when using our approach, and Section 11 concludes and discusses possible future work.

2 Related Work

Several enhancements to the original gradient descent algorithm have been proposed. These include adding a "momentum" term to the update rule [29], and "adaptive gradient" methods such as RMSProp[32], and Adam[14], which combines RMSProp and AdaGrad[5]. These methods have seen widespread use in deep neural networks[19, 33, 1].

Recently, there has been a lot of work on finding novel ways to adaptively change the learning rate. These have both theoretical [21] and intuitive, em-

[1] https://github.com/yrahul3910/adaptive-lr-dnn

pirical [26, 24] backing. These works rely on non-monotonic scheduling of the learning rate. [24] argues for cyclical learning rates. Our proposed method also yields a non-monotonic learning rate, but does not follow any predefined shape.

Our work is also motivated by recent works that theoretically show that stochastic gradient descent is sufficient to optimize over-parameterized neural networks, making minimal assumptions [35, 4]. Our aim is to mathematically identify an optimal learning rate, rejecting the notion that only small learning rates must be used, and then experimentally show the validity of our claims.

We also emphasize here that we discuss extensions to our framework, and apply it to other optimization algorithms; few papers explore these algorithms, choosing instead to only focus on SGD.

3 Theoretical framework

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[2]. 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

[2]Note that we loosely use the term "weights" to refer to both the weight matrices and the biases.

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.

4 Regression

For regression, we use the least-squares cost function. Further, we assume that there is only one output node. That is,

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

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,

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

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

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

The inverse of this, therefore, can be set as the learning rate for gradient descent.

5 Binary classification

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

$$E(\mathbf{z}^{[L]}) = -\frac{1}{m}\left(\mathbf{y}\log g(\mathbf{z}^{[L]}) + (1-\mathbf{y})\log(1-g(\mathbf{z}^{[L]}))\right) \tag{10}$$

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

$$\max_{i,j}\left|\frac{\partial E}{\partial w_{ij}^{[L]}}\right| = \max_{j}\left|\frac{\partial E}{\partial z_{j}^{[L]}}\right| \cdot K_z \tag{11}$$

Then, we have

$$\begin{aligned}
\frac{\partial E}{\partial \mathbf{z}^{[L]}} &= -\frac{1}{m}\left(\frac{\mathbf{y}}{g(\mathbf{z}^{[L]})}g(\mathbf{z}^{[L]})(1-g(\mathbf{z}^{[L]})) - \frac{1-\mathbf{y}}{1-g(\mathbf{z}^{[L]})}g(\mathbf{z}^{[L]})(1-g(\mathbf{z}^{[L]}))\right) \\
&= -\frac{1}{m}\left(\mathbf{y}(1-g(\mathbf{z}^{[L]})) - (1-\mathbf{y})g(\mathbf{z}^{[L]})\right) \\
&= -\frac{1}{m}\left(\mathbf{y} - \mathbf{y}g(\mathbf{z}^{[L]}) - g(\mathbf{z}^{[L]}) + \mathbf{y}g(\mathbf{z}^{[L]})\right) \\
&= -\frac{1}{m}\left(\mathbf{y} - g(\mathbf{z}^{[L]})\right)
\end{aligned} \tag{12}$$

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

$$\frac{\partial^2 E}{\partial \mathbf{w}_{ij}^{[L]2}} = \frac{1}{m}g(\mathbf{z}^{[L]})(1-g(\mathbf{z}^{[L]}))a_j^{[L-1]} \tag{13}$$

Setting (13) 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 (12), we get

$$\max_{j}\left|\frac{\partial E}{\partial z_{j}^{[L]}}\right| = \frac{1}{2m} \tag{14}$$

Using (14) in (11),

$$\max_{i,j}\left|\frac{\partial E}{\partial w_{ij}^{[L]}}\right| = \frac{K_z}{2m} \tag{15}$$

6 General cross-entropy loss function

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

$$E(\mathbf{a}^{[L]}) = -\frac{1}{m}\sum_{j=1}^{k}[\mathbf{y}=j]\log\mathbf{a}^{[L]} \tag{16}$$

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

$$\frac{\partial E}{\partial \mathbf{a}^{[L]}} = -\frac{1}{m}\sum_{j=1}^{m}\frac{[\mathbf{y}=j]}{\mathbf{a}^{[L]}} \tag{17}$$

The second part is computed as follows.

$$\begin{aligned}
\frac{\partial a_j^{[L]}}{\partial z_p^{[L]}} &= \frac{\partial}{\partial z_p^{[L]}}\left(\frac{e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}}\right) \\
&= \frac{[p=j]e^{z_j^{[L]}}\sum_{l=1}^{k} e^{z_l^{[L]}} - e^{z_j^{[L]}}\cdot e^{z_p^{[L]}}}{\left(\sum_{l=1}^{k} e^{z_l^{[L]}}\right)^2} \\
&= \frac{[p=j]e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} - \frac{e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}}\cdot\frac{e^{z_p^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} \\
&= \left([p=j]a_j^{[L]} - a_j^{[L]}a_p^{[L]}\right) \\
&= a_j^{[L]}([p=j] - a_p^{[L]})
\end{aligned} \tag{18}$$

Combining (17) and (18) in (5) gives

$$\frac{\partial E}{\partial W_p^{[L]}} = \frac{1}{m}\left(a_p^{[L]} - [\mathbf{y}=p]\right)K_z \tag{19}$$

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 (19) and combining with (6) gives us our desired result:

$$\max_j\left|\frac{\partial E}{\partial W_j^{[L]}}\right| = \frac{k-1}{km}K_z \tag{20}$$

7 A note on regularization

It should be noted that 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.

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

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 $L = \lambda K$.

8 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.

8.1 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

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

8.2 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

1 $K \leftarrow 0$; $S_{\nabla L} \leftarrow 0$;
2 **for** *each iteration* **do**
3 Compute $\nabla_W L$ on mini-batch;
4 $S_{\nabla L} \leftarrow \beta S_{\nabla L} + (1-\beta)(\nabla_W L)^2$;
5 `// Compute the exponentially weighted average of LC`
6 $K \leftarrow \beta K + (1-\beta)\max\left\|(\nabla_W L)^2\right\|$;
7 `// Weight update`
8 $W \leftarrow W - \frac{\sqrt{K}+\epsilon}{\max\|\nabla_W L\|} \cdot \frac{\nabla_W L}{\sqrt{S_{\nabla L}}+\epsilon}$;
9 **end**

8.3 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`
  $K_1 \leftarrow \beta_1 K_1 + (1-\beta_1)\max\|\nabla_W L\|$;
  $K_2 \leftarrow \beta_2 K_2 + (1-\beta_2)\max\left\|(\nabla_W L)^2\right\|$;
  `// 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 $\max\left\|(\nabla_W L)^2\right\|$. For this reason, we tried two different estimates:

- $\left\|(\max \nabla_W L)^2\right\| = \left\|\left(\frac{k-1}{km}K_z + \lambda\|w\|\right)^2\right\|$ – This set the learning rate high (around 4 on CIFAR-10 with DenseNet), and the model quickly diverged.
- $(\max\|\nabla_W L\|)^2 = \frac{(k-1)^2}{k^2m^2}\max K_z^2 + \lambda^2(\max\|w\|)^2 + \frac{2\lambda(k-1)}{km}K_z(\max\|w\|)$ – 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.

8.4 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.

9 Experiments

Below we show the results and details of our experiments on some publicly available datasets. While our results are not state of the art, our 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 [10]. 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.

[3]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 1: Summary of all experiments

Dataset	Architecture	Algorithm	LR Policy	Weight Decay	Valid. Acc.
MNIST	Custom	SGD	Adaptive	None	99.5%
MNIST	Custom	Momentum	Adaptive	None	**99.57%**
MNIST	Custom	Adam	Adaptive	None	99.43%
CIFAR-10	ResNet20 v1	SGD	Baseline	10^{-3}	60.33%
CIFAR-10	ResNet20 v1	SGD	Fixed	10^{-3}	87.02%
CIFAR-10	ResNet20 v1	SGD	Adaptive	10^{-3}	89.37%
CIFAR-10	ResNet20 v1	Momentum	Baseline	10^{-3}	58.29%
CIFAR-10	ResNet20 v1	Momentum	Adaptive	10^{-2}	84.71%
CIFAR-10	ResNet20 v1	Momentum	Adaptive	10^{-3}	89.27%
CIFAR-10	ResNet20 v1	RMSprop	Baseline	10^{-3}	84.92%
CIFAR-10	ResNet20 v1	RMSprop	Adaptive	10^{-3}	86.66%
CIFAR-10	ResNet20 v1	Adam	Baseline	10^{-3}	84.67%
CIFAR-10	ResNet20 v1	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 v2	SGD	Adaptive	10^{-3}	54.29%
CIFAR-100	ResNet164 v2	SGD	Baseline	10^{-4}	26.96%
CIFAR-100	ResNet164 v2	SGD	Adaptive	10^{-4}	**75.99%**
CIFAR-100	ResNet164 v2	Momentum	Baseline	10^{-4}	27.51%
CIFAR-100	ResNet164 v2	Momentum	Adaptive	10^{-4}	75.39%
CIFAR-100	ResNet164 v2	RMSprop	Baseline	10^{-4}	70.68%
CIFAR-100	ResNet164 v2	RMSprop	Adaptive	10^{-4}	70.78%
CIFAR-100	ResNet164 v2	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%[3]

Table 2: 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)	–	–

A summary of our experiments is given in Table 1. DenseNet refers to a DenseNet[12] architecture with $L = 40$ and $k = 12$.

9.1 MNIST

On MNIST, the architecture we used is shown in Table 2. 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%.

Finally, Figure 1 shows the computed learning rate over epochs. Note that unlike the computed adaptive learning rates for CIFAR-10 (Figure 3) and CIFAR-100 (Figure 7), 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 1. The optimizer

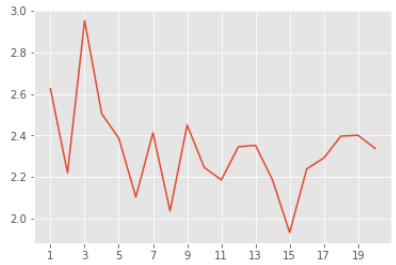


Figure 1: Adaptive learning rate over time on MNIST

achieved its peak validation accuracy after only 8 epochs.

We also used a custom implementation of SGD with momentum (see Appendix A for details), and computed an adaptive learning rate using our AdaMo algorithm. Surprisingly, this outperformed both our adaptive SGD and AutoAdam 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 9.2.

9.2 CIFAR-10

For the CIFAR-10 experiments, we used a ResNet20 v1[10]. 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 [28] 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 [11]. More details can be found in the original ResNet papers [10, 11].

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 2 shows the plots of accuracy score and loss over time. As noted in [25], a horizontal validation loss indicates little overfitting. We achieved a 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.

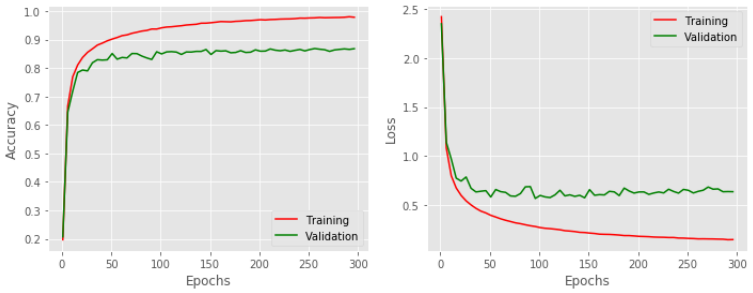
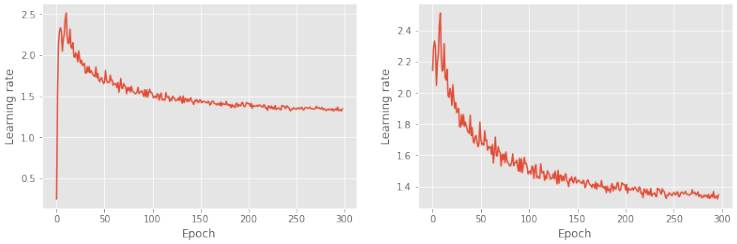
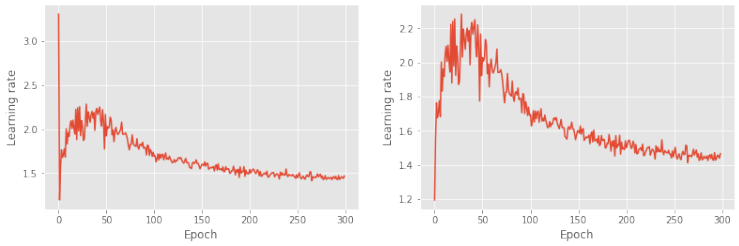


Figure 2: Plot of accuracy score and loss over epochs on CIFAR-10



(a) Learning rate over epochs (b) Learning rate from epoch 2

Figure 3: Adaptive learning rate over time on CIFAR-10



(a) Learning rate over epochs (b) Learning rate from epoch 2

Figure 4: Adaptive learning rate over time on CIFAR-10 using DenseNet

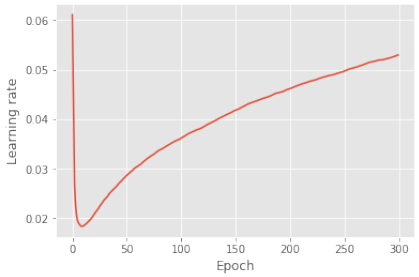


Figure 5: Learning rate over epochs on CIFAR-10 using Adam and DenseNet

Figure 3 shows 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 [26] that large learning rates have a regularization effect.

Figure 4 shows 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-

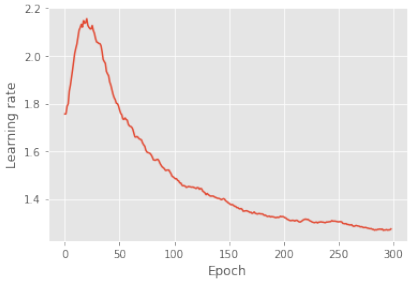


Figure 6: LR over epochs with DenseNet on CIFAR-10 with momentum

10 were when we continued using the learning rate of the first epoch (around 0.05) for all 300 epochs.

Figure 5 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 10.

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 1, for the first experiment using ResNet20 and momentum, we used $x = 0.1$; for the second, we used $x = 1$.

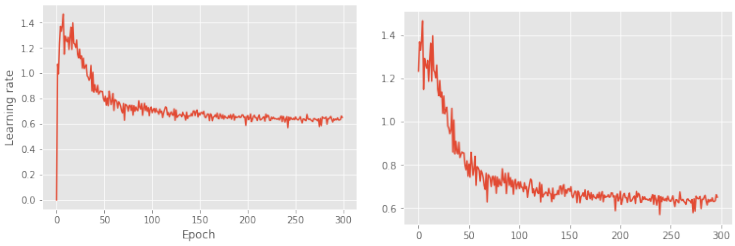
AdaMo also worked well with DenseNets on CIFAR-10. We used $x = 0.5$ for this model. This model crossed 90% validation accuracy before 100 epochs, maintaining a learning rate higher than 1, and was the best among all our models trained on CIFAR-10. This shows the strength of our algorithm. Figure 6 shows the learning rate over epochs for this model.

9.3 CIFAR-100

For the CIFAR-100 experiments, we used a ResNet164 v2 [11]. 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 [11], 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 $x = 1$.



(a) Learning rate over epochs (b) Learning rate from epoch 3

Figure 7: Adaptive learning rate over time on CIFAR-100

Figure 7 shows 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.

9.4 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.

10 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.

11 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 [26], 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.

Acknowledgments

The authors would like to thank the Science and Engineering Research Board (SERB)-Department of Science and Technology (DST), Government of of India for supporting this research. The project reference number is: SERB-EMR/ 2016/005687.

References

[1] Parnia Bahar, Tamer Alkhouli, Jan-Thorsten Peter, Christopher Jan-Steffen Brix, and Hermann Ney. Empirical investigation of optimization algorithms in neural machine translation. *The Prague Bulletin of Mathematical Linguistics*, 108(1):13–25, 2017.

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

[3] Yoshua Bengio, Patrice Simard, Paolo Frasconi, et al. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[4] Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. *arXiv preprint arXiv:1810.02054*, 2018.

[5] 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.

[6] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[7] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[10] 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.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[12] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

[15] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pages 971–980, 2017.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[17] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[18] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[19] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[20] Snehanshu Saha. *Differential Equations: A structured Approach*. Cognella, 2011.

[21] Sihyeon Seong, Yekang Lee, Youngwook Kee, Dongyoon Han, and Junmo 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), 2018.

[22] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[23] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[24] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

[25] Leslie 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.

[26] Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. *arXiv preprint arXiv:1708.07120*, 2017.

[27] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[28] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[29] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[30] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[31] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708, 2014.

[32] 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):26–31, 2012.

[33] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057, 2015.

[34] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[35] Difan Zou, Yuan Cao, Dongruo Zhou, and Quanquan Gu. Stochastic gradient descent optimizes over-parameterized deep relu networks. *arXiv preprint arXiv:1811.08888*, 2018.

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[4]. For ResNets, we used the code from the Examples section of the Keras documentation[5]. The DenseNet implementation we used was from a GitHub repository by Somshubra Majumdar[6]. Finally, our implementation of SGD with momentum is a modified version of the Adam implementation in Keras[7].

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