Optimization of gradient descent

An overview of the most common algorithms used for GD optimization across multiple machine learning settings

Igor Adamski

June 29, 2018

Setting

In machine learning we almost always want to minimize some cost function $L(\boldsymbol{\theta})$

- Let $L(\boldsymbol{\theta})$ be a cost/loss function we want to minimize
- Let $\boldsymbol{\theta}_k \in \mathbb{R}^d$ be the vector of weights of our neural network at iteration k
- Let $\boldsymbol{x}_t = [x_{t,1}, ..., x_{t,n}]$ be a batch/training-set of data at time-step t
- Let $\boldsymbol{g}_k = \nabla_\theta L(\boldsymbol{\theta})$ be the gradient of the loss function, which effectively represents $\boldsymbol{g}_k = (\frac{\partial L(\boldsymbol{\theta})}{\partial \theta_1}, ..., \frac{\partial L(\boldsymbol{\theta})}{\partial \theta_d})^T$. The gradient always points towards the perpendicular direction of greatest ascent.
- Let $\eta \in \mathbb{R}$ be the learning rate/step-size

Convex functions

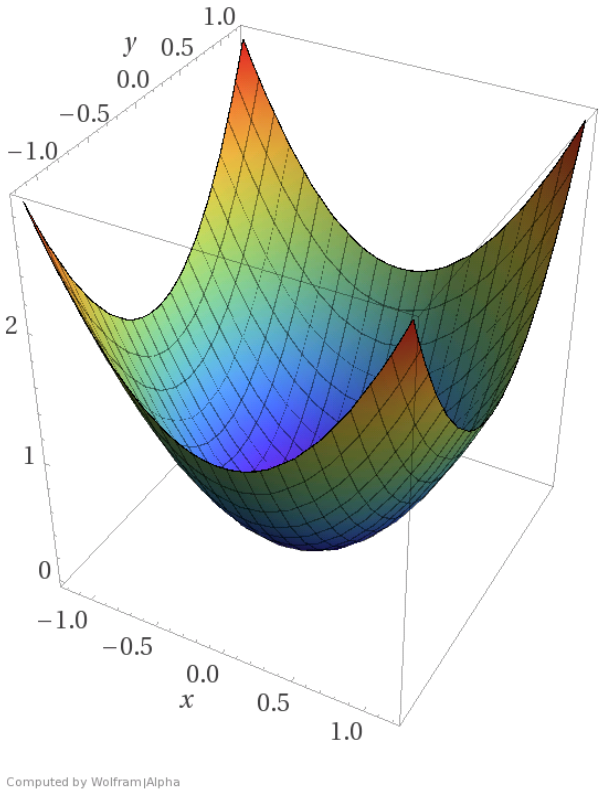


Figure: A convex function

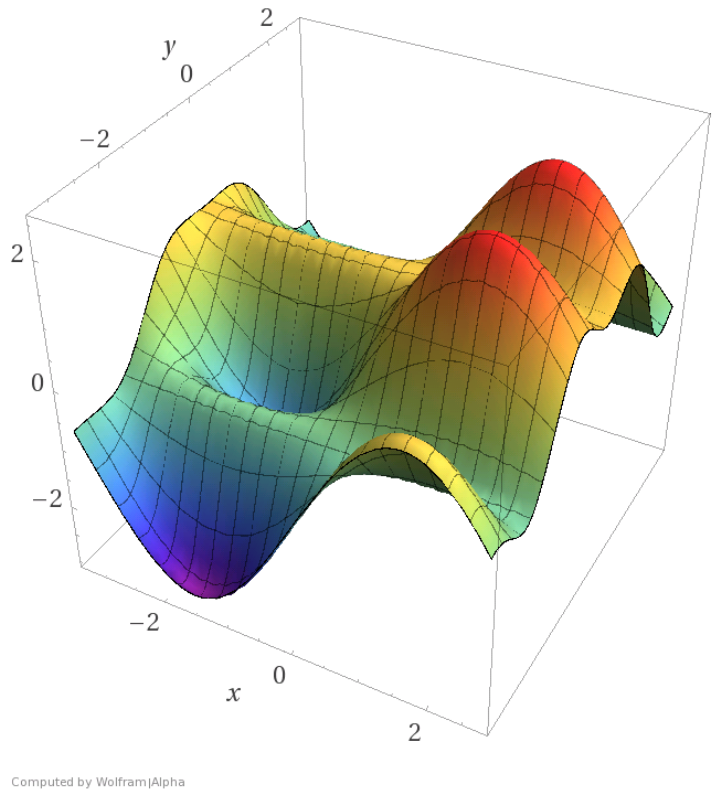


Figure: A non-convex function

3 Variants of gradient descent

$$\mathbb{E}[\nabla L(\boldsymbol{\theta})] = \int \nabla L(\boldsymbol{\theta}, x) p(x) dx \approx \frac{1}{n} \sum_{i=1}^{n} \nabla L(\boldsymbol{\theta}; x^{(i)}) = \nabla L(\boldsymbol{\theta})$$

- **Batch** Performing weight update using the whole batch (entire training-set) of data. Standard error of $\mathbb{E}[\nabla L(\boldsymbol{\theta})]$ estimated by drawing m samples from $p(x)$ is $\frac{\sigma}{\sqrt{m}}$.
- **Online** Performing weight update on a single data point, the loss becomes $L(\boldsymbol{\theta}) = L(\boldsymbol{\theta}, x_k)$. Very noisy. Also not efficient when using GPU.
- **Mini-batch** Performing weight update on a mini-batch of the data, so the loss would be $L(\boldsymbol{\theta}) = \frac{1}{128} \sum_{i=1}^{128} L(\boldsymbol{\theta}, x_i)$. Bigger error but much faster convergence. Mini-batches need to be drawn **randomly** to ensure that we get an unbiased estimator of the gradient.

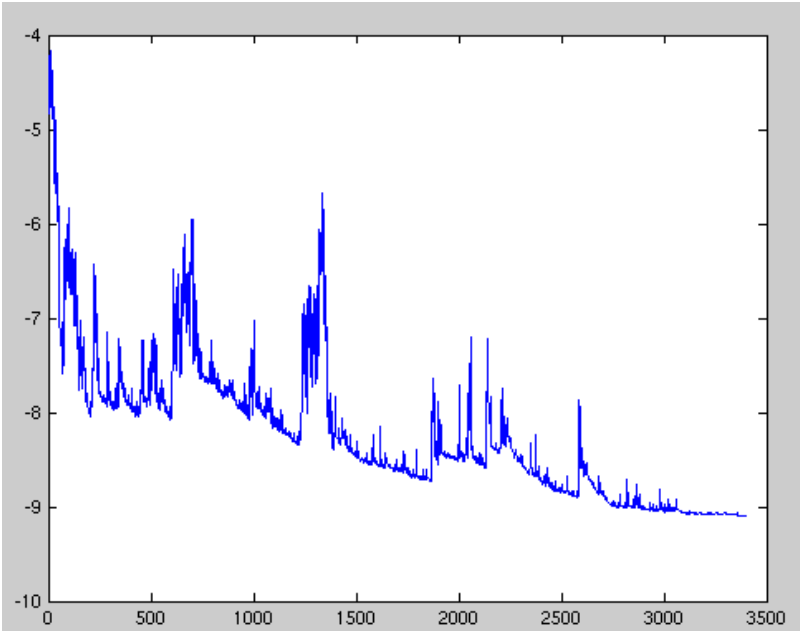


Figure: Loss fluctuation in online SGD

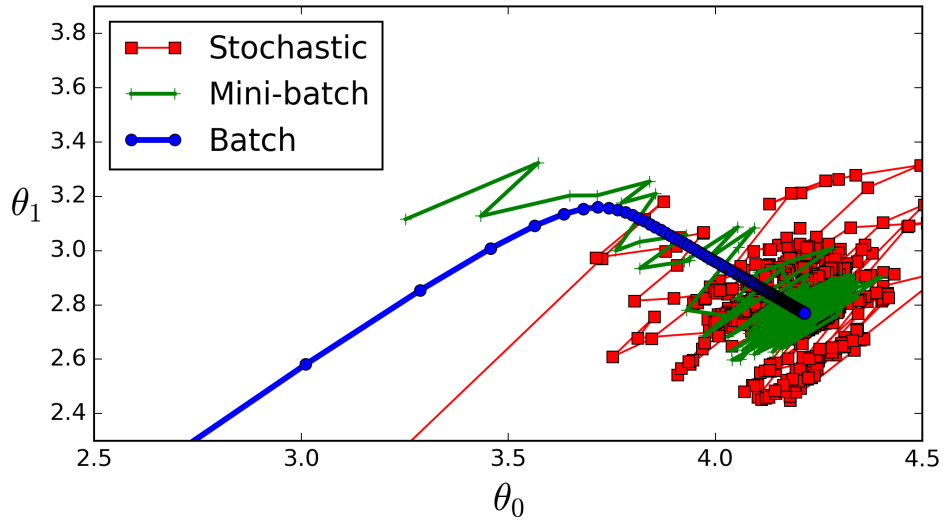


Figure: Convergence comparison for 3 gradientg descent methods

Stochastic gradient descent

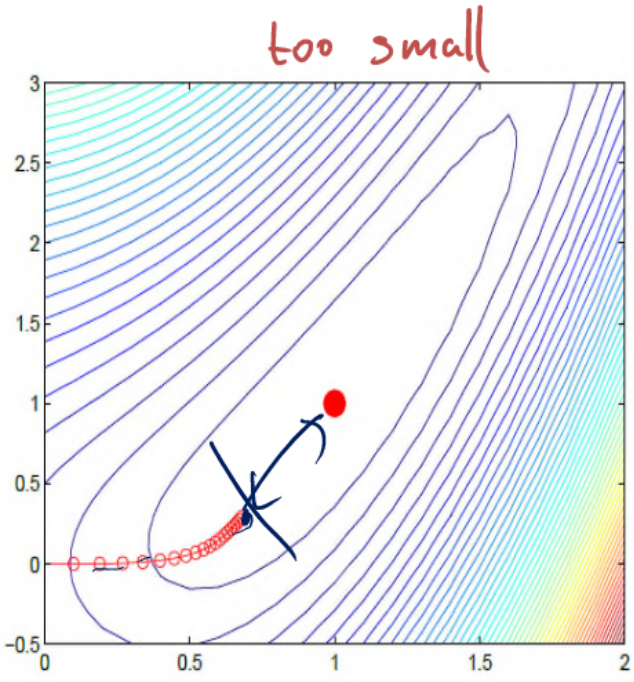
The general update rule for the weights is intuitively:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla L(\boldsymbol{\theta}) = \boldsymbol{\theta}_k - \eta \boldsymbol{g}_k$$

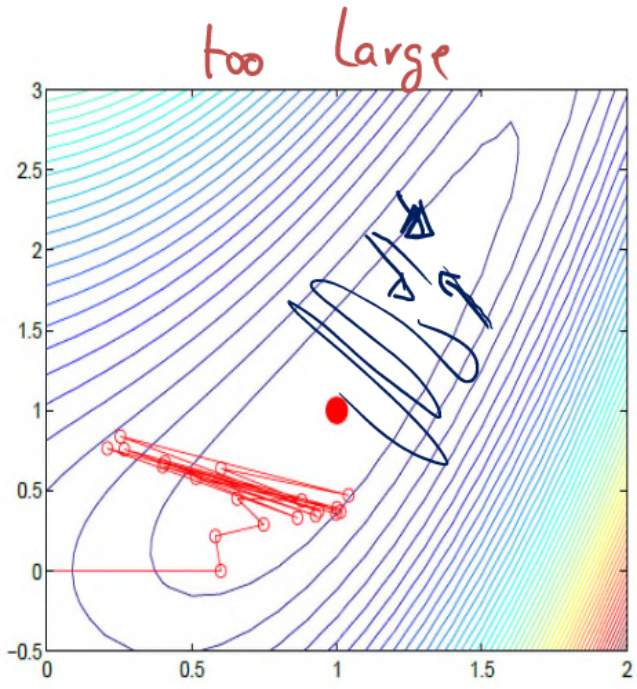
Such an update for online and mini-batch loss functions is called **Stochastic Gradient Descent (SGD)**

Problems with SGD

SGD is the simplest gradient descent algorithm. One of the problems is choosing the value for η.



$\eta = 0.1$.



$\eta = 0.6$.

Problems with SGD

- Choosing the learning rate
- Learning rate annealing (reducing the learning rate constantly during learning) cannot adapt to specific datasets and requires hyperparameters (speed of decay)
- The same learning rate applies to all our weights. If data is huge and sparse then we might not want to update weights to the same extent in every corner of the functions
- SGD has hard time escaping saddle points (for highly non-convex loss functions)

Newton's algorithm

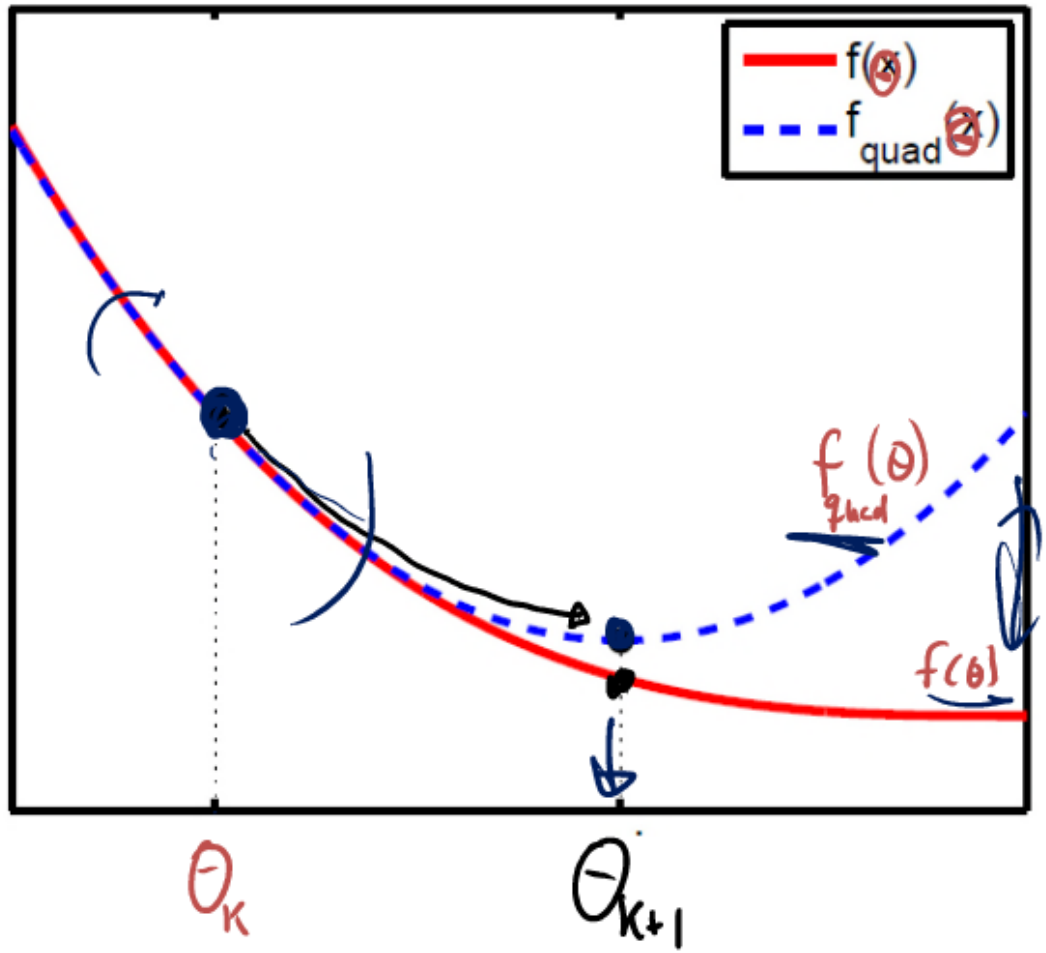
A very efficient method of finding the minimum of $L(\boldsymbol{\theta})$ is by directly looking at the curvature of the loss. Newton's method converges **much faster** than gradient descent; however its rarely used in practice. Why?
The update rule for Newton's method is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - [\boldsymbol{H}_L(\boldsymbol{\theta})]^{-1}\boldsymbol{g}_t$$

where $H_L(\boldsymbol{\theta}) \in \mathbb{R}^{d \times d}$ is the Hessian, so $(H_L(\boldsymbol{\theta}))_{ij} = \dfrac{\partial^2 H_L(\boldsymbol{\theta})}{\partial\theta_i \partial\theta_j}$.

This matrix can be very big and hard to compute.

Newtons method visualisation



- Blue line is the taylor expansion at point θ_t
- Red line is our true function $L(\theta)$

Momentum

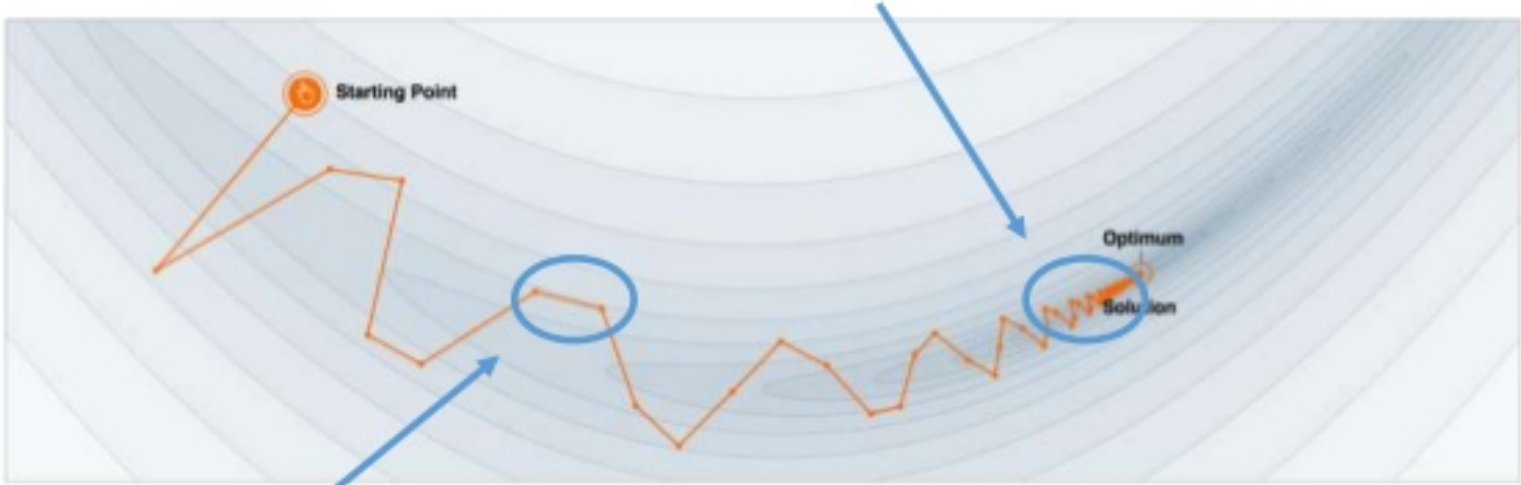
A way to prevent SGD getting stuck in ravines and oscillating near minima. We introduce a memory system so that SGD remembers the past directions. We introduce $v_t = \Delta\boldsymbol{\theta}_t$ so the weight update at iteration t, which we initialize at $v_0 = \mathbf{0}$. Then the update with momentum becomes:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma v_{t-1} - \eta \boldsymbol{g}_t$$

We typically use $\gamma \approx 0.5$ in the beginning of learning as gradients are typically bigger then, and then change to $\gamma \approx 0.9$.

Why Momentum Really Works

The momentum term **reduces updates for dimensions whose gradients change directions.**

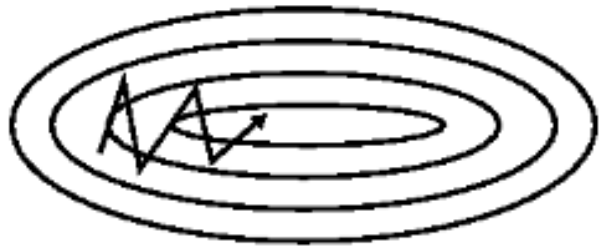


The momentum term **increases for dimensions whose gradients point in the same directions.**

Demo : http://distill.pub/2017/momentum/

Problems

(a) SGD without momentum



(b) SGD with momentum

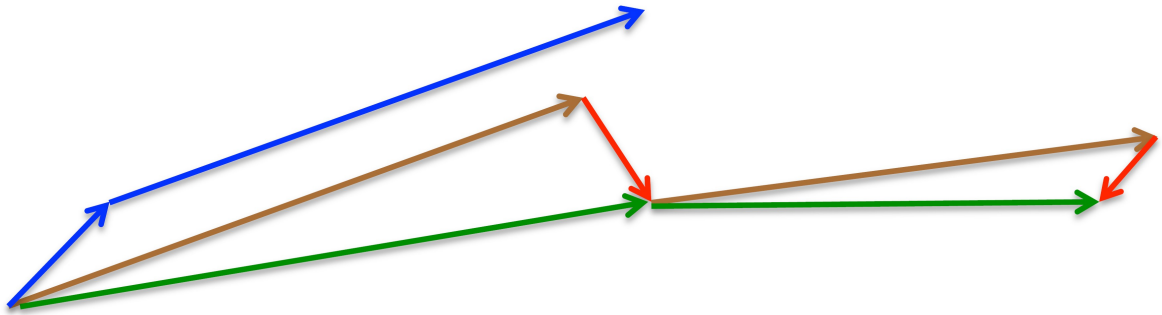
The oscillation problem that SGD had is solved by momentum, but all the weights still share a common η learning rate that cannot properly adapt and act with different magnitudes in different parts of the network.

Nesterov's Accelerated Gradient (NAG)

NAG is a extension of the momentum method. It relies on the principle that we dont want to blindly follow the same directions as we might overshoot beyond the minimum. We want to "slow down" before the "hill" slopes back again. Using momentum we have that $\boldsymbol{\theta}_t - \gamma v_{t-1}$ is an approximation of the $\boldsymbol{\theta}_{t+1}$ (we omit the gradient thus approximation). Then we look to the future and compute the following update:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta L(\boldsymbol{\theta}_t - \gamma v_{t-1})$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - v_t$$

Typical values for γ are as in momentum method.



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

While momentum first goes in direction of gradient (small blue line) and then jumps towards the accumulated gradients (big blue), NAG first jumps towards the previous accumulated gradient, measures the gradient there and corrects. This setup has proven to converge a lot faster.

Adagrad- Adaptive learning algorithm

Adagrad has been introduced to solve the problem of having to manually anneal a learning rate across all parameters. Adagrad stores the values of all previous gradients and thus the learning rate is decreased throughout the process but at different rate for different weights. The update rule for adagrad is:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \frac{\eta}{\sqrt{\sum_{\tau=1}^{k} g_\tau^2 + \epsilon}} \odot g_t$$

where $\odot$ is componentwise multiplication. AdaGrad performs well for some but not all deep learning models, and the learning rate is mostly set $\eta \approx 0.01$

Adaptivity of the learning rate

Let the i^{th} component of $\boldsymbol{\theta}_t$ be $\boldsymbol{\theta}_{t,i}$.
Then, we have that the update for $\boldsymbol{\theta}_{t,i}$ is:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sum_{\tau=1}^{t} g_{\tau,i} + \epsilon} \times g_{t,i}$$

The main advancement over previous gradient descent algorithms is that each parameter has its own unique learning rate, which is especially helpfull when dealoing with sparse gradients.

Pros

- Performs bigger updates in the less frequent directions
- Performs smaller updates in the more frequent directions
- Adapts the learning rate to each weight, which is a property similar to the second order methods as progress along each dimension evens out in time
- Accumulation of gradients in denominator has an annealing property, so we dont have to manually reduce learning rate

Cons

- Very sensitive to initial values of gradients (too large initial values will prevent training)
- The accumulated squared sum in the denominator grows fast and will at some point be large enough to stop learning completely

Adadelta

Adadelta has been developed to overcome the sensitivity on hyperparameters and learning rate decay in Adagrad. It is better to restrict the window of past gradients we are looking at instead of accumulating all the past gradients. Storing ω past gradients is inefficient so we store an **exponentially decaying average of the squared gradients** denoted $\mathbb{E}[g^2]_t$, initialized at $\mathbb{E}[g^2]_0 = \mathbf{0}$ and recursively defined as:

$$\begin{aligned} E[g^2]_t &= \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \\ &= (1-\gamma)[\gamma^{t-1} g_1^2 + ... + \gamma g_{t-1}^2 + g_t^2] \end{aligned}$$

where we pick $\gamma < 1$ so that we have a decaying average.

Then we take the root of that quantity to define:

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

We also wish to get rid of the hand tuned learning rate so that we define recursively a exponentially decaying average of previous updates, initializing at $\mathbb{E}[\Delta\boldsymbol{\theta}^2]_0 = \mathbf{0}$ we continue:

$$\mathbb{E}[\Delta\boldsymbol{\theta}^2]_t = \gamma\mathbb{E}[\Delta\boldsymbol{\theta}^2]_{t-1} + (1-\gamma)\Delta\boldsymbol{\theta}_t^2$$

Now finally the update rule for Adadelta is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\text{RMS}[\Delta\boldsymbol{\theta}]_{t-1}}{\text{RMS}[\boldsymbol{g}^2]_t} \odot \boldsymbol{g}_t$$

where the division is computed componentwise.

Intuition

The intuition behind the $\mathbb{E}[\boldsymbol{g}^2]_t$ is straight-forward as it essentially represents a sum of all squared gradients with the more distant ones discounted and not mattering that much in the sum. It provides a feature similar to Adagrad. The intuitive value of the term $\mathbb{E}[\Delta\boldsymbol{\theta}^2]_t$ is to provide an update in the same units as the weights.

In SGD and Momentum: $\textit{units of } \Delta\theta \propto \textit{units of } g \propto \dfrac{1}{\textit{units of } \theta}$

In Newtons second order method: $\Delta\theta \propto \dfrac{\frac{\partial L}{\partial \theta}}{\frac{\partial^2 L}{\partial \theta^2}} \propto \textit{units of } \theta$

Rearranging the above: $\Delta\theta = \dfrac{\frac{\partial L}{\partial \theta}}{\frac{\partial^2 L}{\partial \theta^2}} \rightarrow \dfrac{1}{\frac{\partial^2 L}{\partial \theta}} = \dfrac{\Delta\theta}{\frac{\partial L}{\partial \theta}}$

Tensorflow implementation

If you have used Tensorflow/Keras before you may have noticed that for Adadelta, the optimizer takes as an input a learning rate η. This is because they essentially do the update:

$$\Delta\boldsymbol{\theta}_t = -\frac{\text{RMS}[\Delta\boldsymbol{\theta}]_{t-1}}{\text{RMS}[\boldsymbol{g}^2]_t} \odot \boldsymbol{g}_t$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \times \Delta\boldsymbol{\theta}$$

And thus if you wish to use the optimizer as stated in the original paper you should use $\eta = 1$

Adadelta

Pros

- We do not have to manually input the learning rate! The learning rate is established by itself closely mimicking the second order approach. It naturally converges to 1 by the end of training.
- Robust to large gradients, noise and architecture choice, because of its ability to remember not only previous gradients but also updates
- Seperate dynamic learning rate per dimension

Cons

- Still have to set hyperparameter γ
- With no proper annealing feature, we could still get trapped in the oscillations

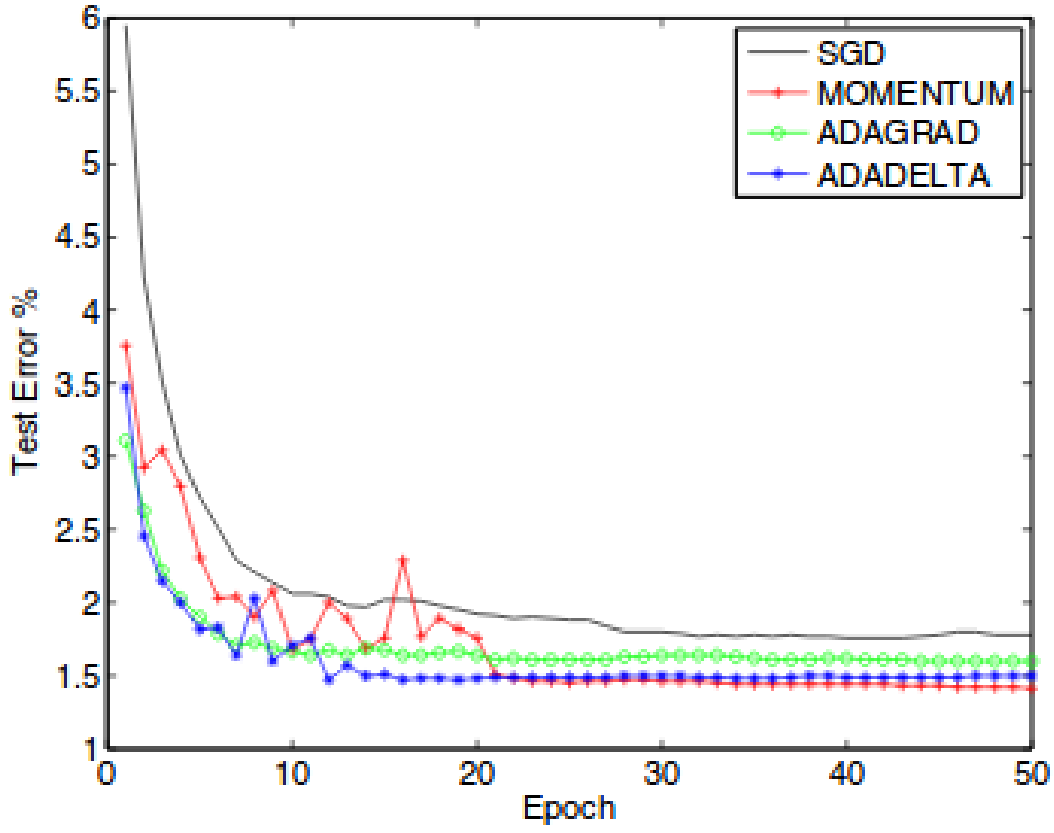


Fig. 1. Comparison of learning rate methods on MNIST digit classification for 50 epochs.

RMSProp

RMSProp is a algorithm proposed by Hinton in his lecture on Coursera class. It essentially stems from the Adadelta method with a chosen $\gamma = 0.9$. Then our decaying average sum of squared gradients become:

$$\begin{aligned} E[g^2]_t &= 0.9 \times E[g^2]_{t-1} + 0.1 \times g_t^2 \\ &= 0.1 \times [0.9^{t-1} g_1^2 + ... + 0.9 g_{t-1}^2 + g_t^2] \end{aligned}$$

and then the update ignores the 'unit correction' of Adadelta simply updating in the following way:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \frac{\eta}{RMS[\boldsymbol{g}^2]_k} \odot \boldsymbol{g}_k$$

Hinton suggests a good value for the learning rate $\eta = 0.001$

Adam

Adam uses the decaying average of the past gradients, being the biased estimate of the **mean** of the gradients, with a recursively defined (initialized at $m_0 = \mathbf{0}$):

$$m_t = \beta_1 \times m_{t-1} + (1 - \beta_1) g_t$$

It also keeps track of the decaying average of past squared gradients (similar to adadelta), which is a biased estimate of the **variance** of past gradients, recursively defined (initialized at $v_0 = \mathbf{0}$):

$$v_t = \beta_2 \times v_{t-1} + (1 - \beta_2) g_t^2$$

Authors give a good starting values at $\beta_1 = 0.9$ and $\beta_2 = 0.999$

Now since the calculated estimates are biased, authors propose a method to correct their biasness by computing:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

And then we can formulate the final update rule which is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Bias correction

Authors observe the bias resulting from initializing both m_t and v_t with zeros at $t = 0$. We look at the derivation for m_t and derivation for v_t is analogous. As before we can write:

$$m_t = \beta_1 \times m_{t-1} + (1 - \beta_1) g_t = (1 - \beta_1) \sum_{k=1}^{t} \beta_1^{t-k} \times g_k$$

We wish to know how $\mathbb{E}[m_t]$ relates to the true first moment $\mathbb{E}[g_t]$. Taking expectation on both sides yields:

$$\begin{aligned}
\mathbb{E}[m_t] &= \mathbb{E}\left[(1 - \beta_1) \sum_{k=1}^{t} \beta_1^{t-k} \times g_k\right] \\
&= \mathbb{E}[g_t] \times (1 - \beta_1) \sum_{k=1}^{t} \beta_1^{t-k} + \xi \\
&= \mathbb{E}[g_t] \times (1 - \beta_1^t) + \xi
\end{aligned}$$

Tensorflow implementation

To be more numerically efficient tensorflow implementation changes the order of computation replacing last few steps with:

$$\eta_t = \eta \times \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \qquad \text{and then}$$

$$\theta_t = \theta_{t-1} - \eta_t \times \frac{m_t}{\sqrt{v_t} + \epsilon}$$

We can see that the bias correction term $\frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \rightarrow 1$ as t increases. In practice, bias correction may disappear after a few minutes of training if we are doing frequent updates.

Pros