Advanced Optimizer

Jue Guo

December 9, 2024

1 Momentum

What happens when performing stochastic gradient descent, i.e., when performing optimization where only a noisy variant of the gradient is available. In particular, we noticed that for noisy gradients we need to be extra cautious when it comes to choosing the learning rate in the face of noise.

- If we decrease it too rapidly, convergence stalls.
- if we are too lenient, we fail to converge to a good enough solution since noise keeps on driving us away from optimality.

1.1 Leaky Averages

We can use minibatch SGD as a means for accelerating computation. It also had the nice side-effect that averaging gradients reduced the amount of variance. The minibatch stochastic gradient descent can be calculated by:

$$\mathbf{g}_{t,t-1}=\partial_{\mathbf{w}}\frac{1}{|\mathcal{B}_t|}\sum_{i\in\mathcal{B}_t}f\left(\mathbf{x}_i,\mathbf{w}_{t-1}\right)=\frac{1}{|\mathcal{B}_t|}\sum_{i\in\mathcal{B}_t}\mathbf{h}_{i,t-1}$$

- To keep the notation simple, here we used $\mathbf{h}_{i,t-1}=\partial_{\mathbf{w}}f\left(\mathbf{x}_i,\mathbf{w}_{t-1}\right)$ as the stochastic gradient descent for sample i using the weights updated at time $t-1$.

It would be nice if we could benefit from the effect of variance reduction even beyond averaging gradients on a minibatch. One option to accomplish this task is to replace the gradient computation by a "leaky average":

$$\mathbf{v}_t=\beta\mathbf{v}_{t-1}+\mathbf{g}_{t,t-1}$$

for some $\beta\in(0,1)$. This effectively replaces the instantaneous gradient by one that is been averaged over multiple *past* gradients. $\mathbf{v}$ is called velocity. It accumulates past gradients similar to how a heavy ball rolling down the objective function landscape integrates over past forces. To see what is happening in more detail let's expand $\mathbf{v}_t$ recursively into

$$\mathbf{v}_t=\beta^2\mathbf{v}_{t-2}+\beta\mathbf{g}_{t-1,t-2}+\mathbf{g}_{t,t-1}=\ldots,=\sum_{\tau=0}^{t-1}\beta^\tau\mathbf{g}_{t-\tau,t-\tau-1}$$

Large β amounts to a long-range average, whereas small β amounts to only a slight correction relative to a gradient method.

- The new gradient replacement no longer points into the direction of steepest decscent on a particular instance any longer but rather in the direction of a weighted average of past gradients. *This allows us to realize most of the benefits of averaging over a batch without the cost of actually computing the gradients on it.*

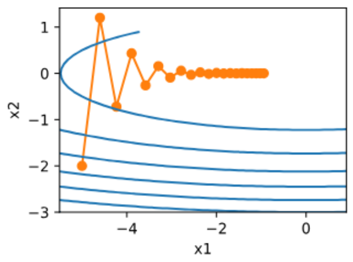
The above reasoning formed the basis for what is known as *accelerated* gradient methods, such as gradients with momentum.

1.2 An ill-conditioned Problem

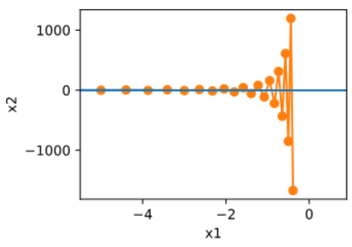
Let's look at a significantly less pleasant objective function:

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$$

knowing that f has its minimum at $(0,0)$. This function is *very* flat in the direction of x_1. We pick a learning rate of 0.4 using gradient descent.



(a) Gradient Descent with lr=0.4



(b) Gradient Descent with lr=0.6

By construction, the gradient in the x_2 direction is much higher and changes much more rapidly than in the horizontal x_1 direction. Thus we are stuck between two undesirable choices: if we pick a small learning rate we ensure that the solution does not diverge in the x_2 direction but we are saddled with slow convergence in the x_1 direction. Conversely, with a large learning rate we progress rapidly in the x_1 direction but diverge in x_2. Even after a slight increase in learning rate from 0.4 to 0.6. Convergence in the x_1 direction improves but the overall solution quality is much worse.

1.3 The Momentum Method

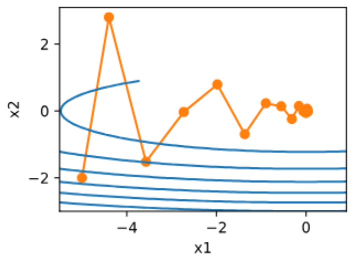
Looking at the optimization trace above we might intuit that averaging gradients over the past would work well.

- After all, in the x_1 direction this will aggregate well-aligned gradients, thus increasing the distance we cover with every step.
- Conversely, in the x_2 direction where gradients oscillate, an aggregate gradient will reduce step size due to oscillations that cancel each other out.

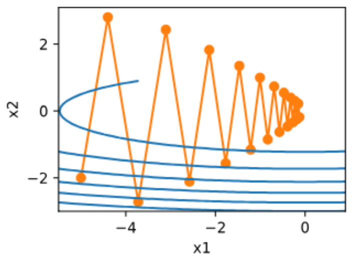
Using $\mathbf{v}_t$ instead of the gradient $\mathbf{g}_t$ yields the following update equations:

$$\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1}$$
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t$$

Note that for $\beta = 0$ we recover regular gradient descent.



(a) lr=0.6; $\beta = 0.5$



(b) lr=0.6; $\beta = 0.25$

1.3.1 Effective Sample Weight

Recall that $\mathbf{v}_t = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}$. In the limit the terms add up to $\sum_{\tau=0}^{\infty} \beta^\tau = \frac{1}{1-\beta}$. In other words, rather than taking a step of size η in gradient descent or stochastic gradient descent we take a step of size $\frac{\eta}{1-\beta}$ while at the same time, dealing with a potentially much better behaved descent direction. These are two benefits in one.

1.4 Theoretical Analysis

So far the $2D$ example of $f(x) = 0.1x_1^2 + 2x_2^2$ seemed rather contrived. We will now see that this is actually quite representative of the types of problem one might encounter, at least in the case of minimizing convex quadratic objective functions.

1.4.1 Quadratic Convex Functions

Consider the function

$$h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} + b.$$

This is a general quadratic function. For positive definite matrices $\mathbf{Q} \succ 0$,i.e., for matrices with positive eigenvalues this has a minimizer at $\mathbf{x}^* = -\mathbf{Q}^{-1}\mathbf{c}$ with minimum value $b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$. Hence we can rewrite h as

$$h(\mathbf{x}) = \frac{1}{2}\left(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c}\right)^\top \mathbf{Q}\left(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c}\right) + b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}.$$

This shows that the function is centered at the minimizer $\mathbf{x}^* = -\mathbf{Q}^{-1}\mathbf{c}$, and the quadratic term describes the "bowl-like" shape of the function around this point. The gradient is given by $\partial_{\mathbf{x}} h(\mathbf{x}) = \mathbf{Q}\left(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c}\right)$. That is, it is given by the distance between $\mathbf{x}$ and the minimizer, multiplied by $\mathbf{Q}$. Consequently also the velocity is a linear combination of terms $\mathbf{Q}\left(\mathbf{x}_t - \mathbf{Q}^{-1}\mathbf{c}\right)$.

Since $\mathbf{Q}$ is positive definite, we can decompose it into its eigensystem:

$$\mathbf{Q} = \mathbf{O}^\top \mathbf{\Lambda} \mathbf{O}$$

where:

- $\mathbf{O}$ is an orthogonal matrix of eigenvectors of $\mathbf{Q}$
- $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues of $\mathbf{Q}$.

Let's define a shifted variable. We shift the variable $\mathbf{x}$ to center the function around the minimizer $\mathbf{X}^*$:

$$\mathbf{x} - \mathbf{x}^* = \mathbf{x} + \mathbf{Q}^{-1}\mathbf{c}$$

This shift re-centers the function so that the quadratic form is now expressed as deviations from the minimizer $\mathbf{x}^*$. Now, we perform the **change of variables**. The idea is to rotate the variable $\mathbf{x} - \mathbf{x}^*$ into the **eigenvector basis** of $\mathbf{Q}$. This allows us to express the quadratic form in a simplified way.

$$\mathbf{z} = \mathbf{O}\left(\mathbf{x} - \mathbf{x}^*\right) = \mathbf{O}\left(\mathbf{x} + \mathbf{Q}^{-1}\mathbf{c}\right)$$

Then substitute it into the quadratic form, we have

$$h(\mathbf{z}) = \frac{1}{2}\mathbf{z}^\top \mathbf{\Lambda}\mathbf{z} + b'$$

Here $b' = b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$. Since $\mathbf{O}$ is only an orthogonal matrix this does not perturb the gradients in a meaningful way. Expressed in terms of $\mathbf{z}$ gradient descent becomes

$$\mathbf{z}_t = \mathbf{z}_{t-1} - \mathbf{\Lambda}\mathbf{z}_{t-1} = (\mathbf{I} - \mathbf{\Lambda})\mathbf{z}_{t-1}$$

The important fact in this expression is that gradient descent does not mix between different eigenspaces. That is, when expressed in terms of the eigensystem of $\mathbf{Q}$ the optimization problem proceeds in a coordinate-wise manner. This also holds for

$$\begin{aligned}\mathbf{v}_t &= \beta \mathbf{v}_{t-1} + \mathbf{\Lambda} \mathbf{z}_{t-1} \\ \mathbf{z}_t &= \mathbf{z}_{t-1} - \eta \left(\beta \mathbf{v}_{t-1} + \mathbf{\Lambda} \mathbf{z}_{t-1}\right) \\ &= (\mathbf{I} - \eta \mathbf{\Lambda}) \mathbf{z}_{t-1} - \eta \beta \mathbf{v}_{t-1}\end{aligned}$$

In doing this we just proved the following theorem: *gradient descent with and without momentum for a convex quadratic function decomposes into coordinate-wise optimization in the direction of the eigenvectors of the quadratic matrix.*

2 Other Optimizers

Let's begin by considering learning problems with features that occur infrequently.

2.1 Sparse Features and Learning Rates

Imagine that we are training a language model. To get good accuracy we typically want to decrease the learning rate as we keep on training, usually at a rate of $\mathcal{O}\left(t^{-\frac{1}{2}}\right)$ or slower. Now consider a model training on sparse features, i.e., features that occur only infrequently. This is common for natural language, e.g., it is a lot less likely that we will see the word *preconditioning* than learning. However, it is also common in other areas such as computational advertising and personalized collaborative filtering. After all, there are many things that are of interest only for a small number of people.

Parameters associated with infrequent features only receive meaningful updates whenever these features occur.

- Given a decreasing learning rate we might end up in a situation where the parameters for common features converge rather quickly to their optimal values, whereas for infrequent features we are still short of observing them sufficiently frequently before their optimal values can be determined.

In other words, the learning rate either decreases too slowly for frequent features or too quickly for infrequent ones.

A possible hack to redress this issue would be to count the number of times we see a particular feature and to use this as a clock for adjusting learning rates.

- That is, rather than choosing a learning rate of the form $\eta = \frac{\eta_0}{\sqrt{t+c}}$ we could use $\eta_i = \frac{\eta_0}{\sqrt{s(i,t)+c}}$. Here $s(i,t)$ counts the number of nonzeros for feature i that we have observed up to time t.

This is actually quite easy to implement at no meaningful overhead. However, it fails whenever we do not quite have sparsity but rather just data where the gradients are often very small and only rarely large. After all, *it is unclear where one would draw the line between something that qualifies as an observed feature or not.*

Adagrad by Duchi et al. addresses this by replacing the rather crude counter $s(i,t)$ by an aggregate of the squares of previously observed gradients. In particular, it uses $s(i,t+1) = s(i,t) + \left(\partial_i f(\mathbf{x})\right)^2$ as a means to adjust the learning rate. This has two benefits:

1. We no longer need to decide just when a gradient is large enough.
2. It scales automatically with the magnitude of the gradients. Coordinates that routinely correspond to large gradient scale down significantly, whereas others with small gradient receive a much more gentle treatment.

In practice this leads to a very effective optimization procedure for computational advertising and related problems. *But this hides some of the additional benefits inherent in Adagrad that are best understood in the context of preconditioning.*

2.2 Algorithm: Adagrad

In short, despite the complex mathematical explanation, adagrad adapts the learning rate to the parameters, performing smaller updates (or low learning rates) for parameters associated with frequently occuring features, and larger updates (or high learning rates) for parameters associated with infrequent features.

Preconditioning Convex optimization problems are a good framework for analyzing algorithms because they offer theoretical guarantees. We start with a quadratic optimization problem:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{c}^\top \mathbf{x} + b$$

where,

- $\mathbf{x}$ is the vector of parameters we want to optimize,
- $\mathbf{Q}$ is a positive definite matrix, meaning it has positive eigenvalues, and
- $\mathbf{c}$ is a constant vector.

To simplify the problem we use **eigendecomposition** of the matrix $\mathbf{Q}$:

$$\mathbf{Q} = \mathbf{U}^\top \mathbf{\Lambda} \mathbf{U}$$

where:

- $\mathbf{U}$ is an orthogonal matrix containing the eigenvectors of $\mathbf{Q}$
- $\mathbf{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\mathbf{Q}$.

This eigendecomposition allows us to rewrite the quadratic function in a simpler form. Next, we change the variable $\mathbf{x}$ to a new variable $\overline{\mathbf{x}}$ by multiplying $\mathbf{x}$ with $\mathbf{U}$:

$$\overline{\mathbf{x}} = \mathbf{U}\mathbf{x}, \overline{\mathbf{c}} = \mathbf{U}\mathbf{c}$$

This transforms the problem into the following form:

$$f(\mathbf{x}) = \bar{f}(\overline{\mathbf{x}}) = \frac{1}{2}\overline{\mathbf{x}}^\top \mathbf{\Lambda}\overline{\mathbf{x}} + \overline{\mathbf{c}}^\top \overline{\mathbf{x}} + b$$

Now, instead of working with the matrix $\mathbf{Q}$, we are working with the diagonal matrix $\mathbf{\Lambda}$, which makes the problem much easier because each term in the quadratic function is independent.

Then we want to minimize the transformed problem, the minimizer of the transformed problem $\bar{f}(\overline{\mathbf{x}})$ is:

$$\overline{\mathbf{x}}^* = -\mathbf{\Lambda}^{-1}\overline{\mathbf{c}}$$

and the minimum value is:

$$f\left(\overline{\mathbf{x}}^*\right) = -\frac{1}{2}\overline{\mathbf{c}}^\top \mathbf{\Lambda}^{-1}\overline{\mathbf{c}} + b$$

At this point, the problem is simplified significantly because we're dealing with a diagonal matrix $\mathbf{\Lambda}$, and each coordinate of $\overline{\mathbf{x}}$ can be solved independently.

Now, if we slightly change $\mathbf{c}$, we expect only slight changes in the minimizer. However, the sensitivity of the minimizer depends on the eigenvalues in $\mathbf{\Lambda}$. Specifically:

- For large eigenvalues $\mathbf{\Lambda}_i$, changes in $\bar{x}_i$ will be small.
- For small eigenvalues $\mathbf{\Lambda}_i$, changes in $\bar{x}_i$ can be large.

The condition number κ is defined as the ratio between the largest and smallest eigenvalues:

$$\kappa = \frac{\mathbf{\Lambda}_1}{\mathbf{\Lambda}_d}$$

If κ is large, the problem is ill-conditioned, meaning small changes in the input can lead to large changes in the output. This makes the optimization problem harder to solve accurately.

Preconditioning aims to "fix" the problem by transforming the space to make the condition number smaller. The idea is to rescale the problem so that all eigenvalues are equal, which makes the problem easier to solve.

1. **Exact Preconditioning**: Theoretically, this can be achieved by transforming $\mathbf{x}$ to a new coordinate system $\mathbf{z} = \mathbf{\Lambda}^{1/2}\mathbf{U}\mathbf{x}$. Doing so we rescales the problem so that in the new coordinate system, the quadratic term becomes:

$$\mathbf{x}^\top \mathbf{Q}\mathbf{x} = \mathbf{z}^\top \mathbf{I}\mathbf{z} = \|\mathbf{z}\|^2$$

   In this coordinate system, the matrix $\mathbf{Q}$ has been transformed into the identity matrix $\mathbf{I}$, where where all eigenvalues are equal to 1. However, calculating the eigendecomposition of $\mathbf{Q}$ can be computationally expensive (often more costly than solving the problem itself).

2. **Approximate Preconditioning**: Instead of computing the full eigendecomposition, a cheaper approximation can be used. One approach is to use the diagonal entries of $\mathbf{Q}$ to precondition the problem:

$$\tilde{\mathbf{Q}} = \operatorname{diag}^{-1/2}(\mathbf{Q})\mathbf{Q}\operatorname{diag}^{-1/2}(\mathbf{Q})$$

   This transformation ensures that the diagonal entries of $\tilde{\mathbf{Q}}$ are all equal to 1, which helps reduce the condition number. It simplifies the problem in cases where the problem is "axis-aligned," i.e., when the largest and smallest values in each direction are significantly different.

We are dealing with a quadratic optimization problem in a transformed coordinate system. To find the gradient $\partial_{\overline{\mathbf{x}}}\bar{f}(\overline{\mathbf{x}})$, we take the derivative of this quadratic function with respect to $\overline{\mathbf{x}}$. Therefore,

$$\partial_{\overline{\mathbf{x}}}\bar{f}(\overline{\mathbf{x}}) = \mathbf{\Lambda}\overline{\mathbf{x}} + \overline{\mathbf{c}}$$

Let $\overline{\mathbf{x}}_0$ denote the minimizer of $\bar{f}(\overline{\mathbf{x}})$. We can calculate $\overline{\mathbf{x}}_0$ by setting the gradient to zero:

$$\mathbf{\Lambda}\overline{\mathbf{x}}_0 + \overline{\mathbf{c}} = 0$$

Solving for $\overline{\mathbf{x}}_0$:

$$\overline{\mathbf{x}}_0 = -\mathbf{\Lambda}^{-1}\overline{\mathbf{c}}$$

Now, substitute this expression for $\overline{\mathbf{x}}_0$ into the gradient equation:

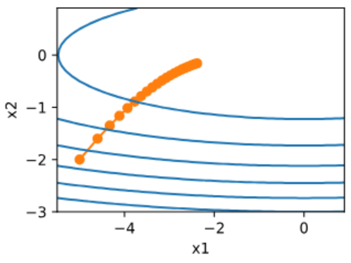
$$\partial_{\overline{\mathbf{x}}}\bar{f}(\overline{\mathbf{x}}) = \mathbf{\Lambda}\overline{\mathbf{x}} + \overline{\mathbf{c}} = \mathbf{\Lambda}\overline{\mathbf{x}} - \mathbf{\Lambda}\overline{\mathbf{x}}_0 = \mathbf{\Lambda}\left(\overline{\mathbf{x}} - \overline{\mathbf{x}}_0\right)$$

Let's formalize the discussion from above. We use the variable $\mathbf{s}_t$ to accumulate past gradient variance as follows.

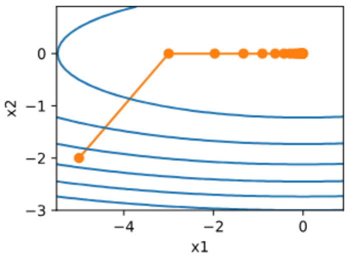
$$\begin{aligned} \mathbf{g}_t &= \partial_{\mathbf{w}} l\left(y_t, f\left(\mathbf{x}_t, \mathbf{w}\right)\right) \\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2 \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t \end{aligned} \tag{1}$$

The algorithm, which adapts the learning rate based on the history of the gradients. The idea is that gradient magnitude gives us information about the curvature of the objective function. Adagrad accumulates the squared gradients over time.

However, from figure 4a due to the cumulative effect of $\boldsymbol{s}_t$, the learning rate continuously decays, so the independent variable does not move as much during later stages of iteration.



(a) adagrad, lr=0.4



(b) adagrad, lr=2

Figure 3: Adagrad; Effect of Different Learning Rate

2.3 RMSProp

One of the key issues in Adagrad is that the learning rate decreases at a predefined schedule of effectively $\mathcal{O}\left(t^{-\frac{1}{2}}\right)$. While this is generally appropriate for convex problems, it might not be ideal for nonconvex ones, such as those encountered in deep learning. Yet, the coordinate-wise adaptivity of Adagrad is highly desirable as a preconditioner.

Tieleman and Hinton (2012) proposed the RMSProp algorithm as a simple fix to decouple rate scheduling from coordinate-adaptive learning rates. The issue is that Adagrad accumulates the squares of the gradient $\mathbf{g}_t$ into a state vector $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$. As a result $\mathbf{s}_t$ keeps on growing without bound due to the lack of normalization, essentially linearly as the algorithm converges.

One way of fixing this problem would be to use $\mathbf{s}_t/t$. For reasonable distributions of $\mathbf{g}_t$ this will converge. Unfortunately it might take a very long time until the limit behavior starts to matter since the procedure remembers the full trajectory of values. An alternative is to use a leaky average in the same way we used in the momentum method, i.e., $\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2$ for some parameter $\gamma > 0$. Keeping all other parts unchanged yields RMSProp.

In short, the main issue with AdaGrad is that it accumulates squared gradients over all time steps, and this accumulation keeps growing. As a result, the learning rate keeps shrinking during training, potentially becoming too small to allow meaningful updates, particularly in deep neural networks where optimization needs to occur over a large number of iterations.

Algorithm RMSProp algorithm computes a moving average of the squared gradients at each step t, denoted by s_t. This is done using an exponential decay (or leaky averaging) factor γ, typically set to a value like 0.9 .

The update rule for s_t is:

$$s_t \leftarrow \gamma s_{t-1} + (1-\gamma) g_t^2$$

where:

- s_t is the running average of squared gradients,
- g_t^2 is the square of the gradient at step t,
- γ controls the decay rate of the squared gradients.

The parameters $\mathbf{x}_t$ are updated based on the gradient g_t, but the learning rate η is scaled down by the square root of s_t to prevent overly large steps. The update rule for the parameters $\mathbf{x}_t$ is:

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot g_t$$

where:

- η is the learning rate,

- ϵ is a small constant (typically 10^{-6}) added for numerical stability (to prevent division by zero),
- $\odot$ denotes element-wise multiplication.

The expression for s_t is expanded to show how it accumulates past squared gradients with a decay factor γ. The expanded form is:

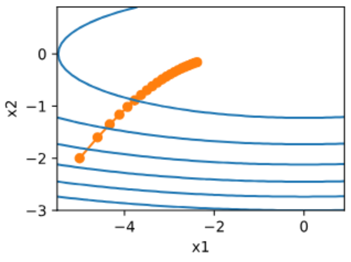
$$s_t = (1-\gamma)g_t^2 + \gamma s_{t-1} = (1-\gamma)\left(g_t^2 + \gamma g_{t-1}^2 + \gamma^2 g_{t-2}^2 + \dots\right)$$

This shows that s_t is a weighted sum of all past squared gradients, where the weight decays exponentially with γ. The sum of the weights in this moving average follows a geometric series:

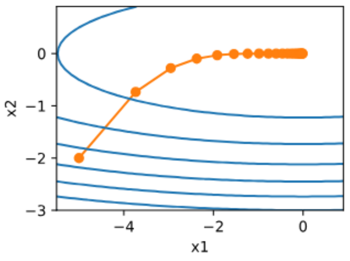
$$1 + \gamma + \gamma^2 + \cdots = \frac{1}{1-\gamma}$$

This normalization ensures that the sum of the weights equals 1 , which gives the correct balance between past and present gradients.

$$\begin{aligned} \mathbf{s}_t &\leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2 \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \tfrac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t \end{aligned} \tag{2}$$



(a) adagrad, lr=0.4



(b) RMSprop, lr=0.4

Figure 4: Adagrad vs RMSprop

2.4 Adadelta

Adadelta starts similarly to AdaGrad and RMSProp by computing a leaky average of the squared gradients. This is done using an exponential moving average with a decay parameter ρ, which controls how fast the memory of past gradients fades away:

$$s_t = \rho s_{t-1} + (1-\rho) g_t^2$$

where s_t accumulates the squared gradients, g_t is the current gradient, and ρ is a decay rate parameter, typically around 0.9 . This equation ensures that recent gradients are more heavily weighted than older ones.

The difference to RMSProp is that we perform updates with the rescaled gradient $\mathbf{g}_t'$, i.e.,

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \mathbf{g}_t'$$

So what is the rescaled gradient $\mathbf{g}_t'$? We can calculate it as follows:

$$\mathbf{g}_t' = \frac{\sqrt{\Delta \mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t$$

where $\Delta \mathbf{x}_{t-1}$ is the leaky average of the squared rescaled gradients $\mathbf{g}_t'$. We initialize $\Delta \mathbf{x}_0$ to be 0 and update it at each step with $\mathbf{g}_t'$, i.e.,

$$\Delta \mathbf{x}_t = \rho \Delta \mathbf{x}_{t-1} + (1-\rho)\mathbf{g}_t'^2$$

and ϵ (a small value such as 10^{-5}) is added to maintain numerical stability.

Summerization of what we learned so far:

- **AdaGrad** accumulates all squared gradients, which can cause the learning rate to decay too quickly, particularly for dense data.
- **RMSProp** uses a moving average of squared gradients to avoid this problem, but still requires a global learning rate.
- **Adadelta** improves on both by introducing adaptive learning rates without requiring a manually defined learning rate parameter. It adapts step sizes based on both the magnitude of past gradients and updates.

2.5 Adam Algorithm

The Adam optimizer combines the key ideas of two other optimization techniques: momentum and RMSProp. By using two moving averages, one for the gradient and one for the squared gradient, Adam adaptively scales the learning rate for each parameter, making it suitable for a wide variety of tasks, especially deep learning applications.

Momentum Term: The first component of Adam is the use of a momentum term, which is an exponentially weighted moving average of past gradients. The purpose of momentum is to smooth out the updates, making the optimization process less sensitive to noise in the gradient. The formula for computing the momentum term at step t is:

$$\mathbf{v}_t = \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1)\mathbf{g}_t$$

In this equation:

- $\mathbf{g}_t$ is the gradient of the loss function with respect to the parameters at step t,
- $\mathbf{v}_t$ is the moving average of past gradients, often referred to as the first moment,
- $\beta_1 \in [0, 1)$ is a hyperparameter that determines how much weight is given to past gradients versus the current gradient. A common value for β_1 is 0.9, which balances responsiveness with smoothness.

The term $(1 - \beta_1)$ ensures that the contribution of the current gradient $\mathbf{g}_t$ is weighted appropriately. This means recent gradients have more influence, while older gradients decay exponentially over time.

Second Moment Estimate: The second major component of Adam is an exponential moving average of the squared gradients, which comes from RMSProp. This term helps to adaptively scale the learning rate for each parameter based on the variability of the gradients. The formula is:

$$\mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2$$

Here:

- $\mathbf{s}_t$ is the moving average of the squared gradients, referred to as the second moment,
- $\beta_2 \in [0, 1)$ is another hyperparameter controlling the decay rate, often set to 0.999,
- $\mathbf{g}_t^2$ represents the element-wise square of the gradient.

The second moment estimate adjusts the step size for parameters with high variance, ensuring that large gradients do not cause excessive parameter updates.

Bias Correction: At the start of optimization, both the momentum term ($\mathbf{v}_t$) and the second moment ($\mathbf{s}_t$) are initialized to zero. This causes their values to be biased towards zero in the early stages of training, especially when β_1^t and β_2^t are small. To address this issue, Adam applies bias correction:

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_1^t}, \quad \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t}$$

Here:

- $\hat{\mathbf{v}}_t$ is the bias-corrected first moment (momentum),
- $\hat{\mathbf{s}}_t$ is the bias-corrected second moment.

These corrections ensure that the moving averages are unbiased, particularly at the beginning of training, allowing for more reliable updates.

Gradient Rescaling: Adam combines the bias-corrected estimates of the momentum and second moment to compute a rescaled gradient. The rescaling normalizes the gradient using the square root of the second moment and a small stabilizing constant ϵ to avoid division by zero. The formula is:

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$$

Here:

- η is the learning rate, controlling the overall step size,
- ϵ is a small constant (e.g., 10^{-8}) added for numerical stability, preventing division by zero.

This rescaled gradient ensures that parameters with high variance gradients take smaller steps, while those with low variance gradients can take relatively larger steps.

Parameter Update: Finally, the parameters are updated by subtracting the rescaled gradient:

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \mathbf{g}_t'$$

This update rule adjusts the parameters based on both the direction (momentum term) and the scale (second moment) of the gradient, leading to smoother and more reliable convergence.

How Adam Differs from AdaGrad and RMSProp

Adam is an evolution of AdaGrad and RMSProp:

- **From AdaGrad:** While AdaGrad adjusts learning rates based on the cumulative sum of squared gradients, it suffers from rapidly decaying learning rates, which can cause the algorithm to stall. Adam mitigates this by using a moving average of squared gradients rather than a cumulative sum, allowing it to adapt without diminishing step sizes prematurely.
- **From RMSProp:** RMSProp introduces a second moment estimate to scale the learning rate based on gradient variance. Adam extends this idea by adding a momentum term, allowing it to leverage the history of gradients for smoother updates.

In addition there is *a variant of Adam*, Yogi improves upon Adam in cases where the second moment estimate $\mathbf{s}_t$ becomes excessively large, which can lead to instability. Yogi modifies the update rule for the second moment to prevent rapid growth:

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \odot \operatorname{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$$

This update allows Yogi to handle scenarios where the gradient variance is overestimated, ensuring better stability during optimization and preventing overly conservative updates.