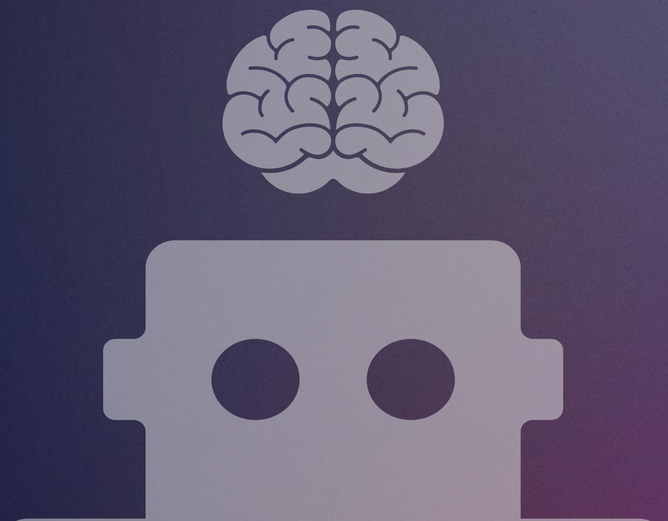
INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

HUAWEI

Optimizers

(for Deep Learning Training)

Prof. Me. Saulo A. F. Oliveira
saulo.oliveira@ifce.edu.br



Optimizers

- There are various optimized versions of gradient descent algorithms. In object-oriented language implementation, different gradient descent algorithms are often encapsulated into objects called optimizers.
- Purposes of the algorithm optimization include but are not limited to:
  - ➡ Accelerating algorithm convergence.
  - ➡ Preventing or jumping out of local extreme values.
  - ➡ Simplifying manual parameter setting, especially the learning rate (LR).
- Common optimizers: **common GD optimizer**, **momentum optimizer**, Nesterov, **AdaGrad**, AdaDelta, **RMSProp**, **Adam**, AdaMax, and Nadam.

01

Optimizing GD algorithm

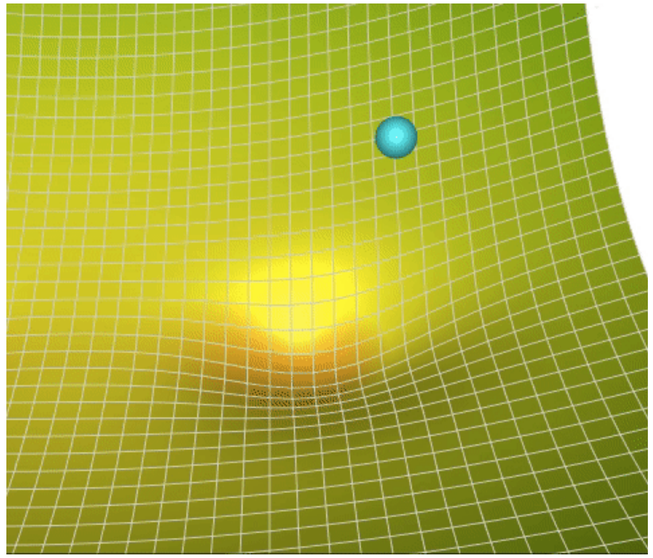
INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

HUAWEI

Vanilla Gradient Descent

- Let's have a quick refresher. In the context of machine learning, the goal of gradient descent is usually to minimize the loss function for a machine learning problem. A good algorithm finds the minimum fast and reliably well (i.e. it doesn't get stuck in local minima, saddle points, or plateau regions, but rather goes for the global minimum).
- The basic gradient descent algorithm follows the idea that the opposite direction of the gradient points to where the lower area is. So it iteratively takes steps in the opposite directions of the gradients. For each parameter theta, it does the following:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla J\left(\boldsymbol{\theta}^{(t)}\right).$$



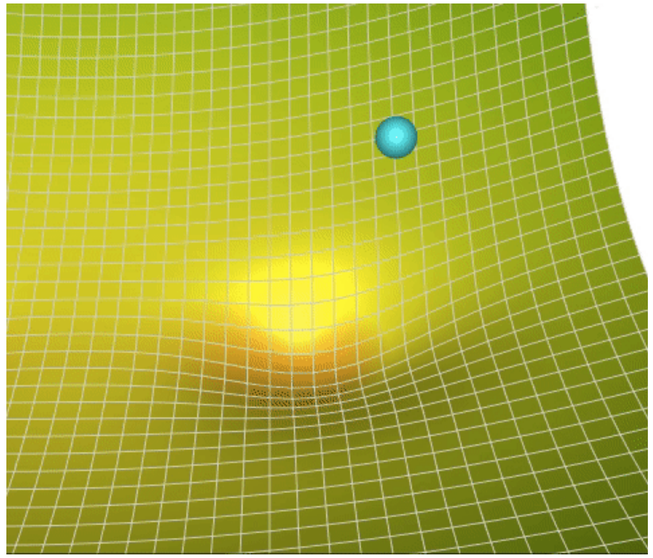
https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-

- Theta is some parameter you want to optimize (for example, the weight of a neuron-to-neuron connection in neural network, the coefficient for a feature in linear regression, etc). There can be thousands of such thetas in an ML optimization setting. Delta is the amount of change for theta after every iteration in the algorithm; the hope is that with each of such change, theta is incrementally getting closer to the optimum.

Vanilla Gradient Descent

- Let's have a quick refresher. In the context of machine learning, the goal of gradient descent is usually to minimize the loss function for a machine learning problem. A good algorithm finds the minimum fast and reliably well (i.e. it doesn't get stuck in local minima, saddle points, or plateau regions, but rather goes for the global minimum).
- The basic gradient descent algorithm follows the idea that the opposite direction of the gradient points to where the lower area is. So it iteratively takes steps in the opposite directions of the gradients. For each parameter theta, it does the following:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla J\left(\boldsymbol{\theta}^{(t)}\right).$$



https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-

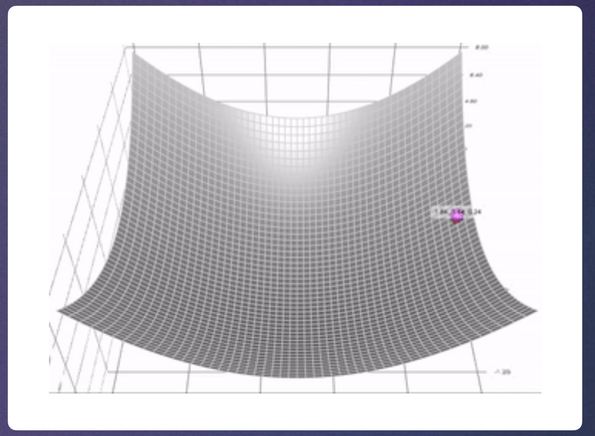
- Theta is some parameter you want to optimize (for example, the weight of a neuron-to-neuron connection in neural network, the coefficient for a feature in linear regression, etc). There can be thousands of such thetas in an ML optimization setting. Delta is the amount of change for theta after every iteration in the algorithm; the hope is that with each of such change, theta is incrementally getting closer to the optimum.

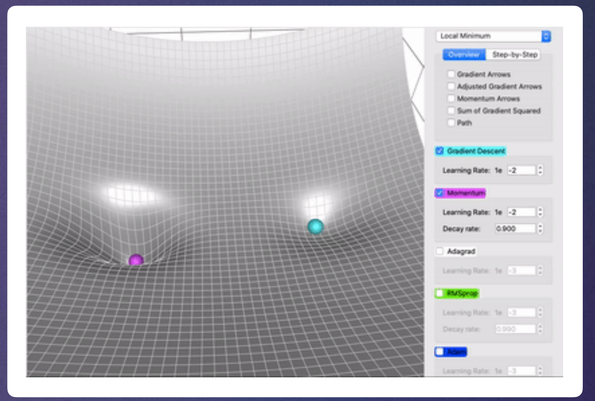
Momentum

- The gradient descent with momentum algorithm (or Momentum for short) borrows the idea from physics. Imagine rolling down a ball inside of a frictionless bowl. Instead of stopping at the bottom, the momentum it has accumulated pushes it forward, and the ball keeps rolling back and forth.
- We can apply the concept of momentum to our vanilla gradient descent algorithm. In each step, in addition to the regular gradient, it also adds on the movement from the previous step. Mathematically, it is commonly expressed as:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla J\left(\boldsymbol{\theta}^{(t)}\right) + \alpha \nabla J\left(\boldsymbol{\theta}^{(t-1)}\right),$$

where α is a constant $(0 \geq \alpha \geq 1)$ called Momentum Coefficient and $\alpha \nabla J\left(\boldsymbol{\theta}^{(t-1)}\right)$ is a momentum term.



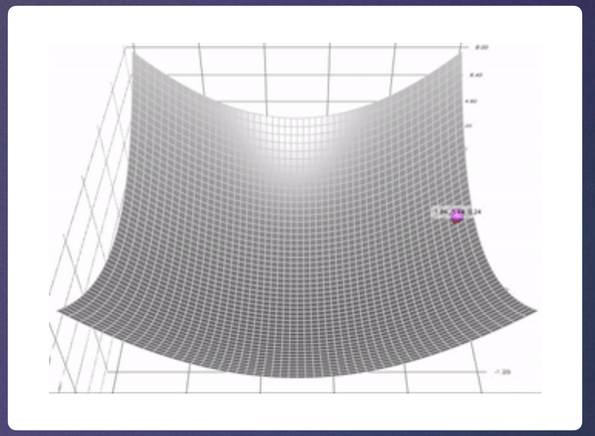


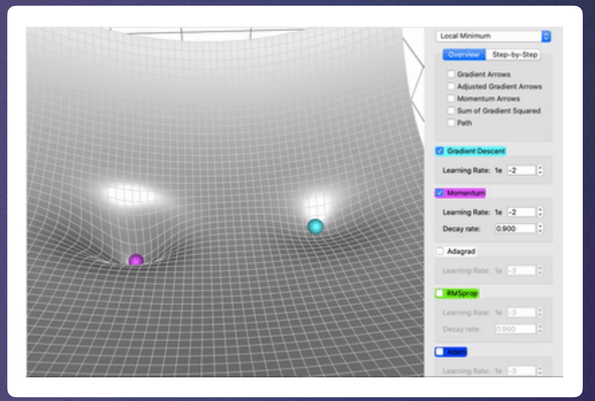
Momentum

- The gradient descent with momentum algorithm (or Momentum for short) borrows the idea from physics. Imagine rolling down a ball inside of a frictionless bowl. Instead of stopping at the bottom, the momentum it has accumulated pushes it forward, and the ball keeps rolling back and forth.
- We can apply the concept of momentum to our vanilla gradient descent algorithm. In each step, in addition to the regular gradient, it also adds on the movement from the previous step. Mathematically, it is commonly expressed as:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla J\left(\boldsymbol{\theta}^{(t)}\right) + \alpha \nabla J\left(\boldsymbol{\theta}^{(t-1)}\right),$$

where α is a constant $(0 \geq \alpha \geq 1)$ called Momentum Coefficient and $\alpha \nabla J\left(\boldsymbol{\theta}^{(t-1)}\right)$ is a momentum term.



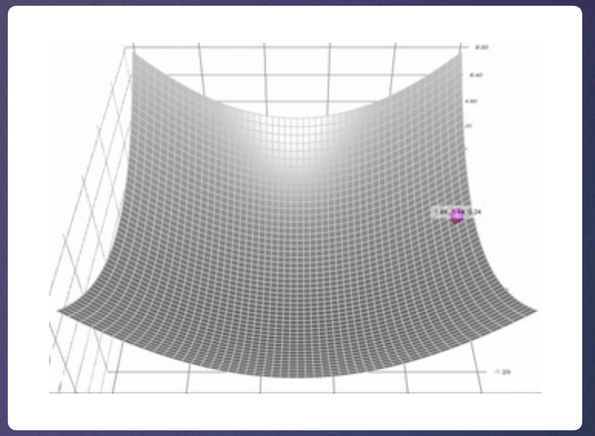


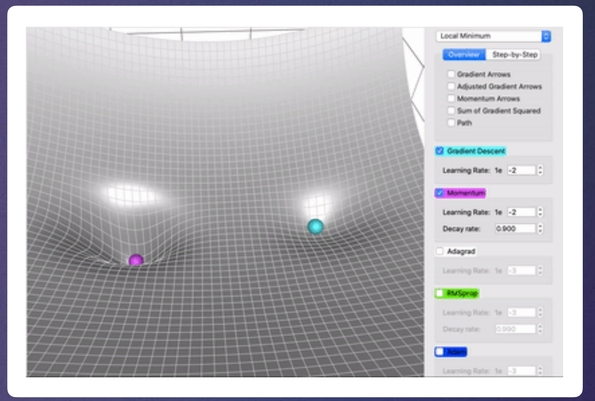
Momentum

- The gradient descent with momentum algorithm (or Momentum for short) borrows the idea from physics. Imagine rolling down a ball inside of a frictionless bowl. Instead of stopping at the bottom, the momentum it has accumulated pushes it forward, and the ball keeps rolling back and forth.
- We can apply the concept of momentum to our vanilla gradient descent algorithm. In each step, in addition to the regular gradient, it also adds on the movement from the previous step. Mathematically, it is commonly expressed as:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla J\left(\boldsymbol{\theta}^{(t)}\right) + \alpha \nabla J\left(\boldsymbol{\theta}^{(t-1)}\right),$$

where α is a constant $(0 \geq \alpha \geq 1)$ called Momentum Coefficient and $\alpha \nabla J\left(\boldsymbol{\theta}^{(t-1)}\right)$ is a momentum term.

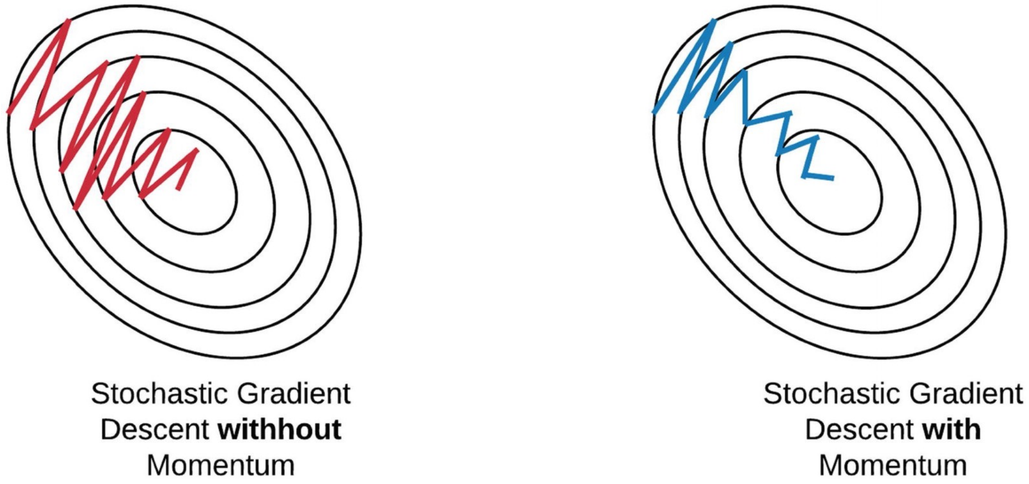




Advantages and Disadvantages of Momentum Optimizer

ADVANTAGES:

- Enhances the stability of the gradient correction direction and reduces mutations.
- In areas where the gradient direction is stable, the ball rolls faster and faster (there is a speed upper limit because $(\alpha < 1)$which helps the ball quickly overshoot the flat area and accelerates convergence.
- A small ball with inertia is more likely to roll over some narrow local extrema.



https://eloquentarduino.github.io/2020/04/stochastic-gradient-descent-on-your-microcontroller/

DISADVANTAGES:

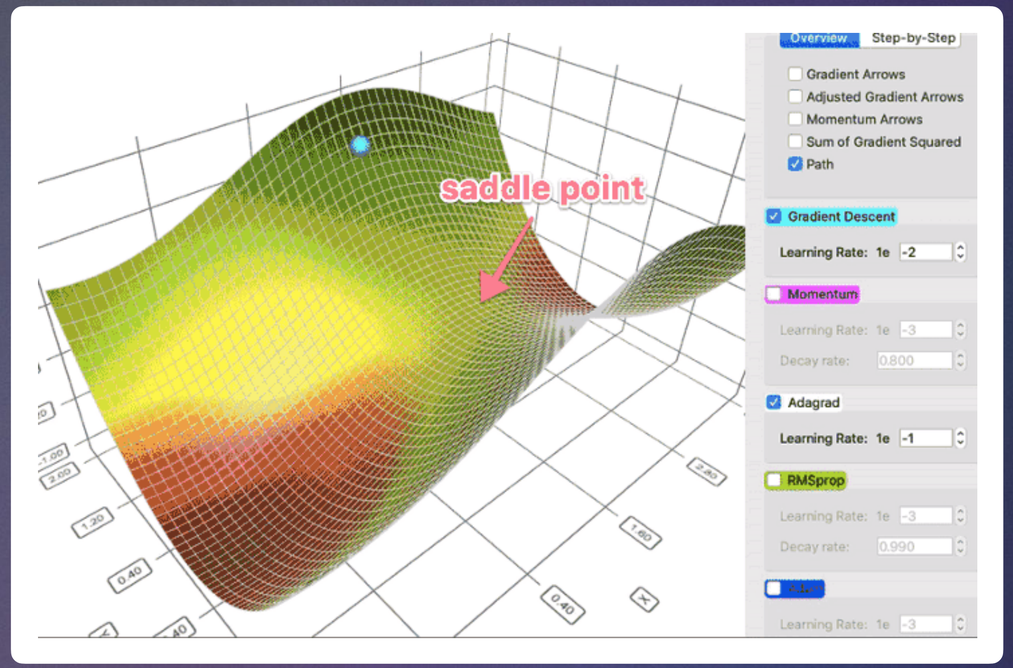
- The learning rate η and momentum α need to be manually set, which often requires more experiments to determine the appropriate value.

AdaGrad (1)

- In ML optimization, some features are very sparse. The average gradient for sparse features is usually small so such features get trained at a much slower rate. One way to address this is to set different learning rates for each feature, but this gets messy fast.
- Instead of keeping track of the sum of gradient like momentum, the **Ada**ptive **Grad**ient algorithm, or AdaGrad for short, keeps track of the sum of gradient squared and uses that to adapt the gradient in different directions. Often the equations are expressed in tensors. I will avoid tensors to simplify the language here. For each dimension:

$$r^{(t)} = r^{(t-1)} + \nabla J\left(\theta^{(t)}\right)^2,$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\varepsilon + \sqrt{r^{(t)}}} \nabla J\left(\theta^{(t)}\right).$$



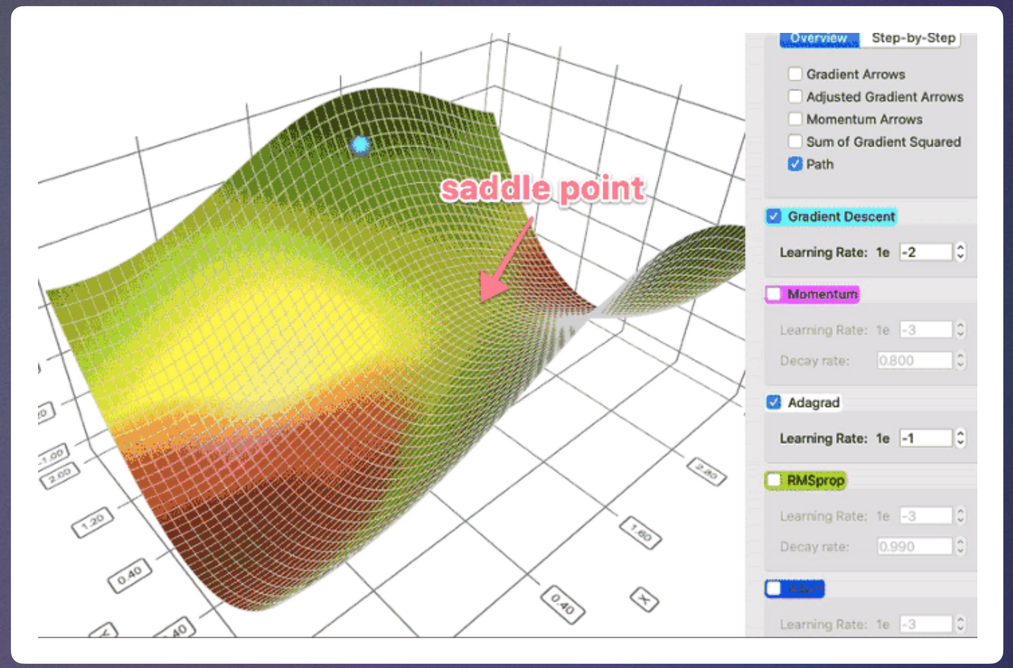
- AdaGrad addresses this problem using this idea: the more you have updated a feature already, the less you will update it in the future, thus giving a chance for the others features (for example, the sparse features) to catch up. the initial value of r is 0, which increases continuously while ε is a small constant, and is set to about 10^{-7} for numerical stability.

AdaGrad (1)

- In ML optimization, some features are very sparse. The average gradient for sparse features is usually small so such features get trained at a much slower rate. One way to address this is to set different learning rates for each feature, but this gets messy fast.
- Instead of keeping track of the sum of gradient like momentum, the **Ada**ptive **Grad**ient algorithm, or AdaGrad for short, keeps track of the sum of gradient squared and uses that to adapt the gradient in different directions. Often the equations are expressed in tensors. I will avoid tensors to simplify the language here. For each dimension:

$$r^{(t)} = r^{(t-1)} + \nabla J\left(\theta^{(t)}\right)^2,$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\varepsilon + \sqrt{r^{(t)}}} \nabla J\left(\theta^{(t)}\right).$$



- AdaGrad addresses this problem using this idea: the more you have updated a feature already, the less you will update it in the future, thus giving a chance for the others features (for example, the sparse features) to catch up. the initial value of r is 0, which increases continuously while ε is a small constant, and is set to about 10^{-7} for numerical stability.

AdaGrad (2)

- The AdaGrad optimization algorithm shows that the r continues increasing while the overall learning rate keeps decreasing as the algorithm iterates. This is because we hope LR to decrease as the number of updates increases. In the initial learning phase, we are far away from the optimal solution to the loss function. As the number of updates increases, we are closer to the optimal solution, and therefore LR can decrease.
- **Pros**:
  - ➡ The learning rate is automatically updated. As the number of updates increases, the learning rate decreases.
- **Cons**:
  - ➡ The denominator keeps accumulating so that the learning rate will eventually become very small, and the algorithm will become ineffective.

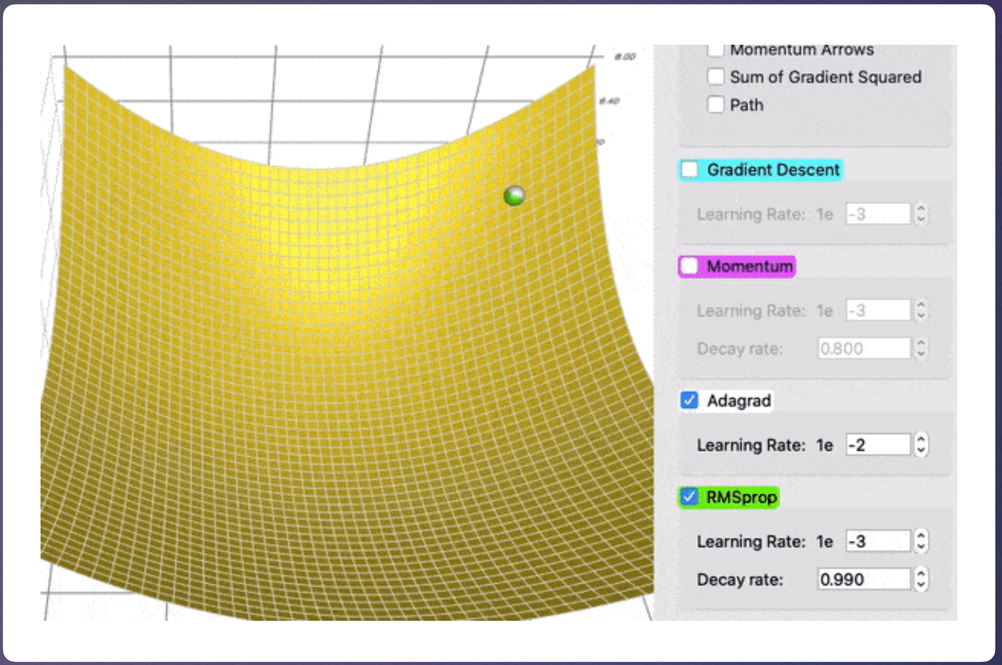
RMSProp

- The problem of AdaGrad, however, is that it is incredibly slow. This is because the sum of gradient squared only grows and never shrinks. RMSProp (for **R**oot **M**ean **S**quare **P**ropagation) fixes this issue by adding a decay factor β:

$$r^{(t)} = \beta r^{(t-1)} + (1-\beta)\nabla J\left(\theta^{(t)}\right)^2,$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\varepsilon + \sqrt{r^{(t)}}}\nabla J\left(\theta^{(t)}\right).$$

- More precisely, the sum of gradient squared is actually the decayed sum of gradient squared.



- The decay rate is saying only recent gradient² matters, and the ones from long ago are basically forgotten. The *sums of gradient squared* for AdaGrad accumulate so fast that they soon become humongous (demonstrated by the sizes of the squares in the animation). On the other hand, thanks to the decay rate, RMSProp is faster than AdaGrad.

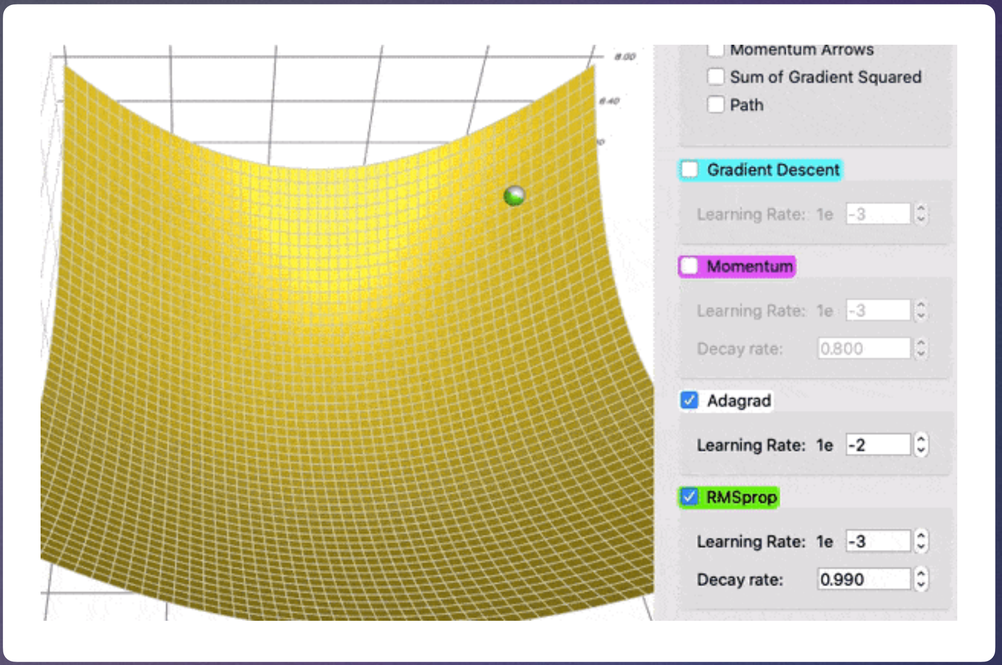
RMSProp

- The problem of AdaGrad, however, is that it is incredibly slow. This is because the sum of gradient squared only grows and never shrinks. RMSProp (for **R**oot **M**ean **S**quare **P**ropagation) fixes this issue by adding a decay factor β:

$$r^{(t)} = \beta r^{(t-1)} + (1-\beta)\nabla J\left(\theta^{(t)}\right)^2,$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\varepsilon + \sqrt{r^{(t)}}}\nabla J\left(\theta^{(t)}\right).$$

- More precisely, the sum of gradient squared is actually the decayed sum of gradient squared.



- The decay rate is saying only recent gradient² matters, and the ones from long ago are basically forgotten. The *sums of gradient squared* for AdaGrad accumulate so fast that they soon become humongous (demonstrated by the sizes of the squares in the animation). On the other hand, thanks to the decay rate, RMSProp is faster than AdaGrad.

Adam

- Last but not least, Adam (short for Adaptive Moment Estimation) takes the best of both worlds of Momentum and RMSProp. Adam empirically works well, and thus in recent years, it is commonly the go-to choice of deep learning problems. Let's take a look at how it works:

$$r^{(t)} = \beta_1 r^{(t-1)} + (1-\beta_1)\nabla J\left(\theta^{(t)}\right), \quad \text{\#Mometum}$$

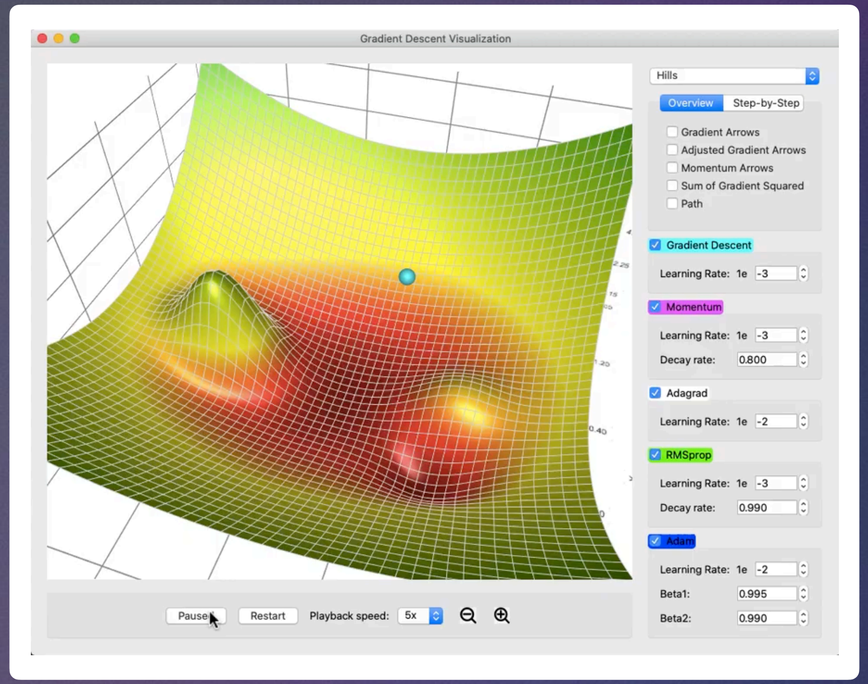
$$m^{(t)} = \beta_2 m^{(t-1)} + (1-\beta_2)\nabla J\left(\theta^{(t)}\right)^2, \quad \text{\#RMSProp}$$

- If $r^{(t)}$ and $m^{(t)}$ are initialized using the zero vector, $r^{(t)}$ and $m^{(t)}$ are close to 0 during the initial iterations, especially when β_1 and β_2 are close to 1. To solve this problem, we employ both $\bar{r}^{(t)}$ and $\bar{m}^{(t)}$:

$$\bar{r}^{(t)} = \frac{r^{(t)}}{1-\beta_1}, \quad \text{and} \quad \bar{m}^{(t)} = \frac{m^{(t)}}{1-\beta_2},$$

- So that, the weight update rule of Adam is as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\varepsilon + \sqrt{r^{(t)}}} m^{(t)}.$$



- Although the rule involves manual tuning of η, β_1 and β_2, the setting is much simpler. According to experiments, the default settings are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ and $\eta = 0.001$. Adam gets the speed from momentum and the ability to adapt gradients in different directions from RMSProp. The combination of the two makes it powerful.

https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c

Adam

- Last but not least, Adam (short for Adaptive Moment Estimation) takes the best of both worlds of Momentum and RMSProp. Adam empirically works well, and thus in recent years, it is commonly the go-to choice of deep learning problems. Let's take a look at how it works:

$$r^{(t)} = \beta_1 r^{(t-1)} + (1-\beta_1)\nabla J\left(\theta^{(t)}\right), \quad \text{\#Mometum}$$

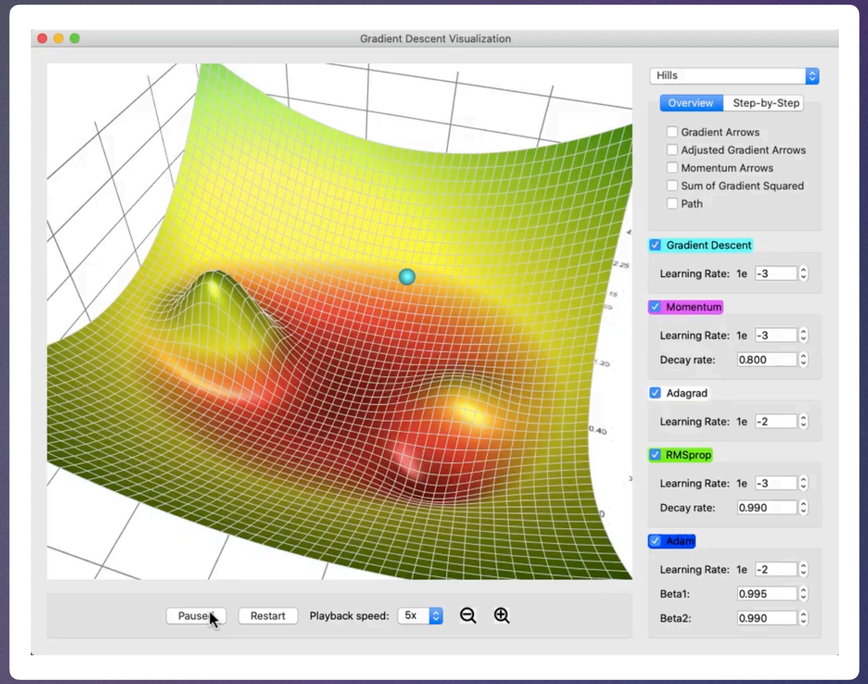
$$m^{(t)} = \beta_2 m^{(t-1)} + (1-\beta_2)\nabla J\left(\theta^{(t)}\right)^2, \quad \text{\#RMSProp}$$

- If $r^{(t)}$ and $m^{(t)}$ are initialized using the zero vector, $r^{(t)}$ and $m^{(t)}$ are close to 0 during the initial iterations, especially when β_1 and β_2 are close to 1. To solve this problem, we employ both $\bar{r}^{(t)}$ and $\bar{m}^{(t)}$:

$$\bar{r}^{(t)} = \frac{r^{(t)}}{1-\beta_1}, \quad \text{and} \quad \bar{m}^{(t)} = \frac{m^{(t)}}{1-\beta_2},$$

- So that, the weight update rule of Adam is as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\varepsilon + \sqrt{r^{(t)}}} m^{(t)}.$$



- Although the rule involves manual tuning of η, β_1 and β_2, the setting is much simpler. According to experiments, the default settings are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ and $\eta = 0.001$. Adam gets the speed from momentum and the ability to adapt gradients in different directions from RMSProp. The combination of the two makes it powerful.

https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c

References

- Lili Jiang. A Visual Explanation of Gradient Descent Methods (Momentum, AdaGrad, RMSProp, Adam). https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c. 2020, Accessed on Feb 2021.
- HUAWEI. Deep Learning Overview. 2020, Accessed on Feb 2021.

INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

HUAWEI

Thank you for your attention!

🙏☺️

Prof. Me. Saulo A. F. Oliveira
saulo.oliveira@ifce.edu.br