Problem 1

A) Stochastic gradient descent + Momentum

SGD

Consider the function of the parameter θ that we want to minimize with k number of samples:

$L(\theta) = \frac{1}{k}\sum_{i=1}^{k} l(y_i, \hat{y}_i; \theta)$

Where y is the correct answer that is present in our dataset, and $\hat{y}$ is the value predicted for our model. The loss function can be of any kind. (cross - entropy loss, MSE etc)

Using raw gradient descent, the $L(\theta)$ is calculated using the formula mentioned above and the θ is estimated using the following equation:

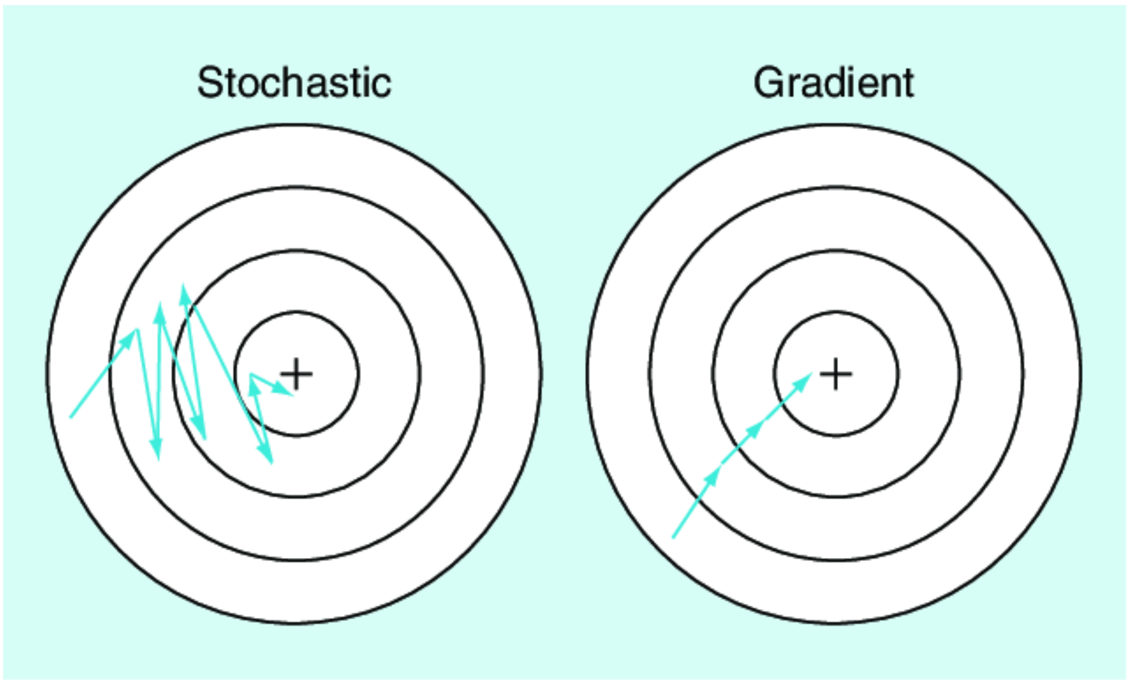
$\theta_{n+1} = \theta_n - \alpha \nabla L(\theta)$

Where α is our learning rate.

After enough iterations, the we'll converge to a local minimum of the function. But **Stochastic gradient descent** brings us a faster way to converge to the local minimum of the function. The main idea behind SGD is that instead of doing each iteration for all of k samples, in each iteration we only calculate $l(y_i, \hat{y}_i; \theta)$ and its gradient, and use the $\theta_{n+1} = \theta_n - \alpha \nabla l(y_i, \hat{y}_i; \theta)$ equation to get to the minimum where i is each time a randomly chosen number from the range of $[1, k]$. (A randomly chosen sample)

SGD VS GD

Because in stochastic gradient descent we only see one sample at a time, we may get bad convergence in that iteration, but on the other hand, we are doing far less computation



As you can see in the above image, the gradient descent has a proper and straight forward convergence into the minumum; While the stochastic gradient descent algorithm look a little bit messy (even though it converges to the minimum after enought iteration)

Pros

- **Faster**: The stochastic gradient descent is a faster algorithm comparing to gradient descent because we're doing less job in each iteration of upading the θ.
- **Less computational load**: Let $g(n)$ be the time complexity of calculating $l(y_i, \hat{y}_i; \theta)$, n be the number of iterations we do to get to the minumum and k be the number of samples. Then the time complexity of stochastic gradient descent is $O(g(n).\, m)$ and the time complexity of gradient descent is $O(g(n).\, m.\, k)$
- **Larger samples compatible**: Because the stochastic gradient is fast, it can be used to large training samples while the gradient descent might not be suitble for larger samples.

Cons:

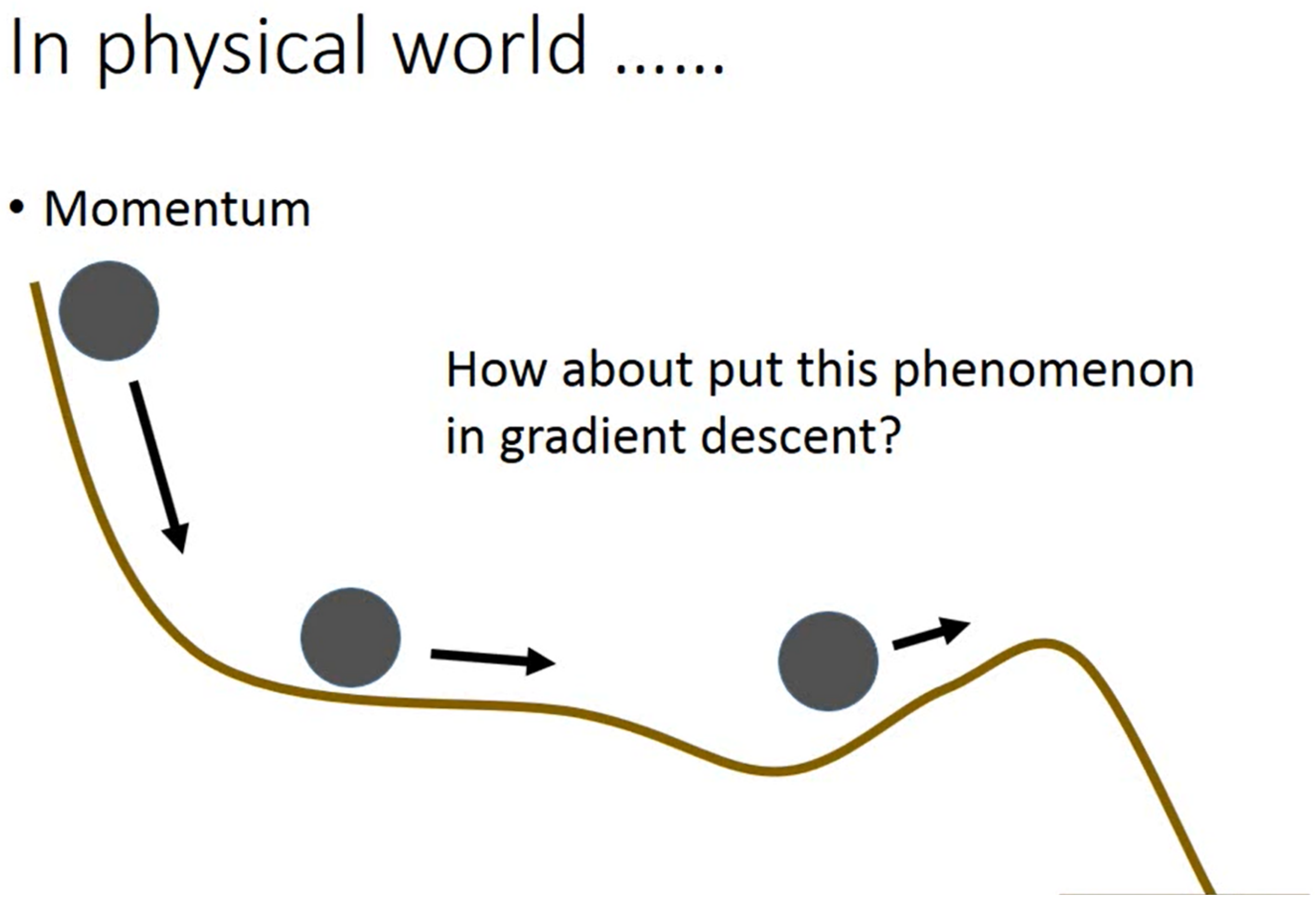
- **Less convergence accuracy**: Because only one sample is seen at each step, the chance of getting to optimal solution is less than getting to optimal solution in gradient descent.

Momentum

Image a ball rolling down a mountain range in $x - y$ plane. There are several low points on the mountain that the ball may get to at the end, but when it gets to a low point for the first time, it doesn't necessarily stop there because the ball has volacity, and because the ball has **momentum**. Momentum of a mass m going with volacity v in physics is defined as follows:

$p = m.\, v$

The higher the momentum, the further the ball will get from the low point that is has just reached and if it could (the momentum is enough and the low point isn't the lowest point in our range) the ball would get to the next low point. (The higher the volacity, the higher the momentum)



When using gradient descent, we can use this problem in physics to avoid getting stuck at local minimum (and eventually get to the next minimum point). Therefor we should define a volcaity for our algorithm to achieve that. But what properties the volacity should have?

1. First of all, the more negative the gradient, the higher the volacity. (remember when the ball gets faster in steeper direction?)
2. At each iteration, the volcaity of that iteration effects the volacity of the next iteration.

Considering these two properties, the volacity at step n can be calculated as follows:

$$v_n = \beta v_{n-1} - \alpha \nabla Loss(\theta)$$

Where β and α are hyperparameters. So how do we determine the value of the parameter θ in step n? Back to the rolling ball example. The x poisition of the ball can be calculated using the following formula:

$$x_{n+1} = x_n + v_x . \Delta t$$

We can use this equation to calculate the parameter:

$$\theta_{n+1} = \theta_n + v_n$$

Pros

- **Escaping local minimum**: Recall from the rolling ball, the ball may escape a local minimum if it has enought momentum getting to that point. Our model can actually do the same if its volcaity is high enough and the loss function allows us to.
- **Faster convergence**: Because the volcaity is a function of the steep in each iteration and the volacity in the previous iteration, the volacity gets higher and higher when the gradient is negative. Which causes the θ parameter to take bigger and bigger steps towards the minimum.

There is another point in momentum that I'll mention in **SGD + momentum** section.

SGD + Momentum

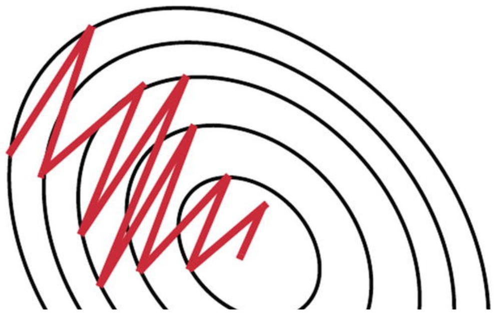
When SGD and momentum are used together, they become a very powerfull optimization technique.

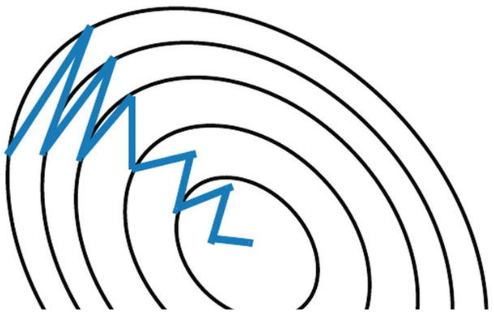
Here's the formula for SGD + momentum:

$$v_n = \beta v_{n-1} - \alpha \nabla l(y_i, \hat{y}_i; \theta)$$

$$\theta_{n+1} = \theta_n + v_n$$

Because the next value of θ depends on the v_n and v_n depends on v_{n-1} and so, the convergence path of our algorithm smoothes out. So when the gradient has a value that would make our model to get further from minumum point in pure SGD, the v_{n-1} helps our model to not lose its mind and get much further from minimum.

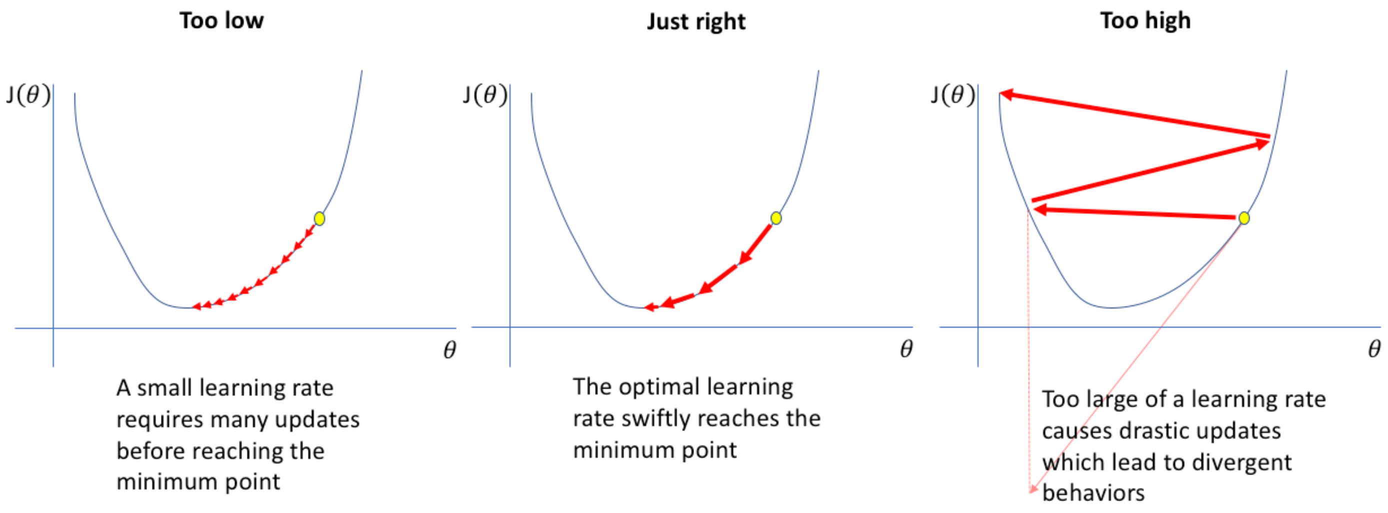




Stochastic Gradient Descent **withhout** Momentum
Stochastic Gradient Descent **with** Momentum

B) AdaGrad

The problem with constant learning rate is that high learning rates cause oscillations (which is the state that the model gets stuck around minimum without actually converging it), and the problem with low learning rates is that even though they have a good accuracy at converging to minimum point, but they take a lost of time.


Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

The AdaGrad technique is a technique to change learning rate value dynamically. At step i it stores a sum of all of the squares of gradients that have been calculated from step 0 to i. This is the formula for it:

Let g_i be ∇l_i, then:

$$G_i = \sum_{t=0}^{i}(g_t)(g_t^T)$$

And the parameter θ is upadted through:

$$\theta_i = \theta_{i-1} - \frac{\alpha}{\sqrt{diag(G_i) + \epsilon}} g_i$$

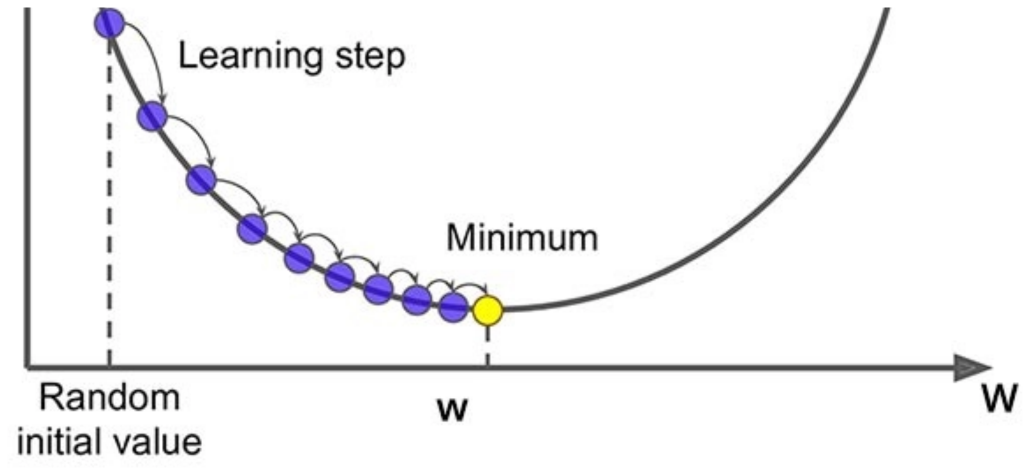
The diagnol entries in G_i is actually the sum of squares of g_t, so we can rewrite the equations like this:

$$G_i = G_{i-1} + g_{i-1}^2$$

$$\theta_i = \theta_{i-1} - \frac{\alpha}{\sqrt{G_i + \epsilon}} g_{i-1}$$

Where ϵ is a small constant, in order to prevent the square root to be 0 or a really small number.

So AdaGrad, keeps a running average of the squares of gradients. And the learning rate is getting smaller and smaller over time.



C) RMSProp

RMSProp is another technique, based on AdaGrad. The difference is that it keeps a *weighted** moving average of the squares of gradients. Let's see how.

This is the equation for the weighted moving average at step i that RMSProp keeps:

$A_i = \gamma . A_{i-1} + (1-\gamma) . (\nabla L_i(\theta))^2$

Let's break that down into a few terms of it to get the pattern:

Let g_i be $\nabla L_i(\theta)$, then:

$A_i = \gamma . A_{i-1} + (1-\gamma) . g_i^2$

$A_0 = A_0$

$A_1 = \gamma . A_0 + (1-\gamma) g_1^2$

$A_2 = \gamma^2 . A_0 + \gamma . (1-\gamma) g_1^2 + (1-\gamma) g_2^2$

$A_3 = \gamma^3 . A_0 + \gamma^2 . (1-\gamma) g_1^2 + \gamma(1-\gamma) g_2^2 + (1-\gamma) g_3^2$

And so on; It seems that we can rewrite the A_n in the following form:

$A_n = \gamma^n . A_0 + \sum_{i=1}^{n} \gamma^{n-i} (1-\gamma) g_i^2$

The $\gamma^n . A_0$ is a constant, so we don't have to worry about it. But the summation is really important. It's summing over all of the G_i^2s of the previous steps while giving more weight to more recent squares of gradients and less weight to older ones. Because the lower the i, the lower the $\gamma^{n-i}(1-\gamma)$ term. $(0 < \gamma < 1)$

So what if we want to give smaller learning rates when the moving average is higher and bigger learning rates when the moving average to lower moving averages? The $\text{learning rate} \propto \frac{1}{\text{size of the gradient}}$. And also we know that $\text{size of the gradient} \propto \sqrt{A_i}$, so all together, we'll get that:

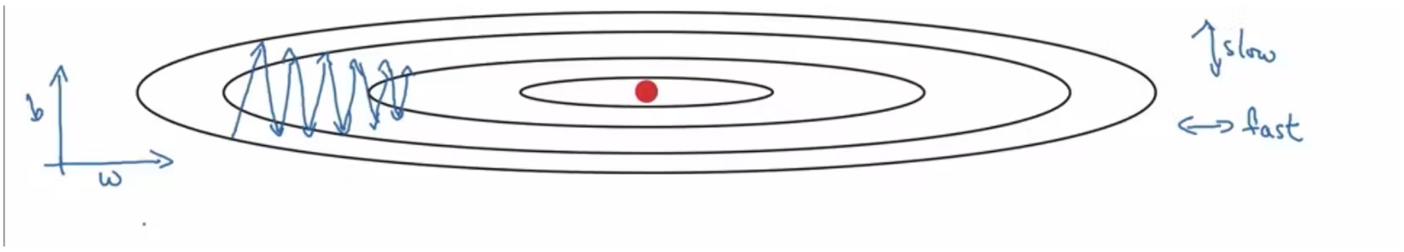
$\text{learning rate} \propto \frac{1}{\sqrt{A_i}}$

Now using this fact to upadte the parameters:

$\theta_{n+1} = \theta_n - \frac{\alpha}{\sqrt{A_i}} . \nabla L_i(\theta)$

Where α is our new constant.

> RMSprop



As you can see the steps get smaller and smaller as we get closer and closer to the minimum point.