

Introduction to Gradient Descent



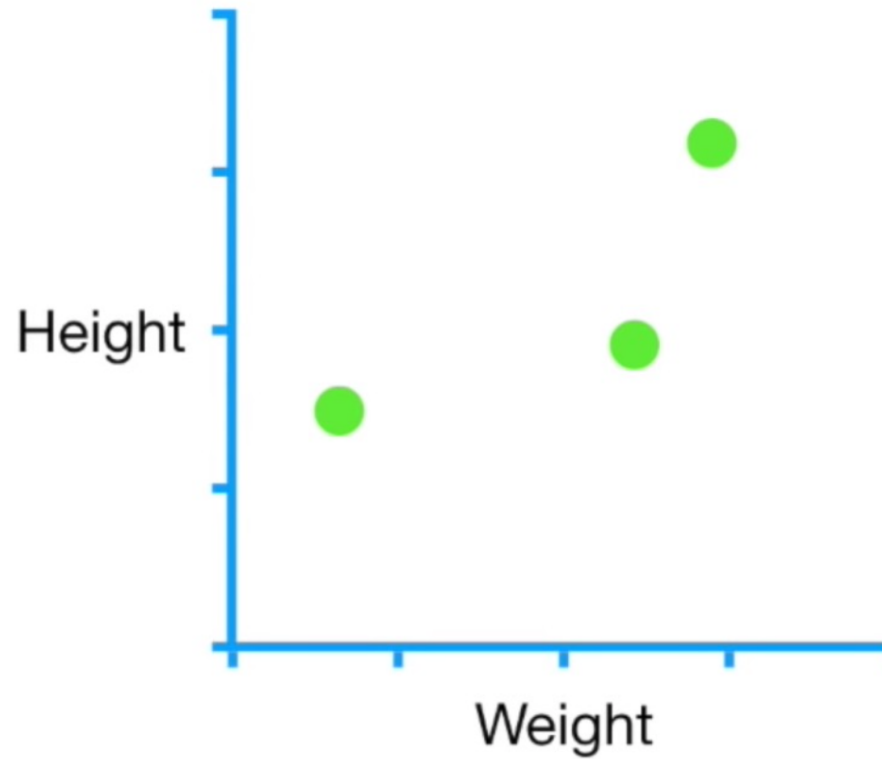
By Alan Grunberg

(with graphs adapted from Le Wagon and StatQuest)

Linear regression with 2 features:
finding the line $Y = mX + b$ that best fits the relationship

m = slope
 b = intercept

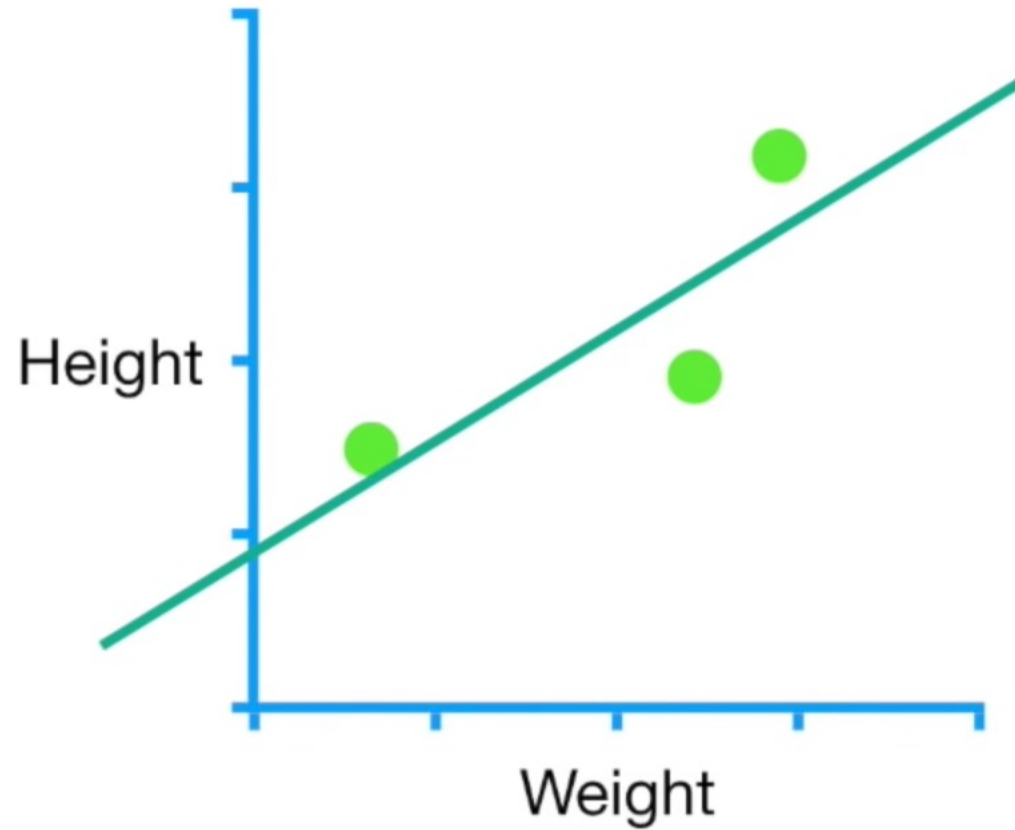
	weight	height
0	0.7	1.5
1	2.4	1.8
2	2.8	3.2



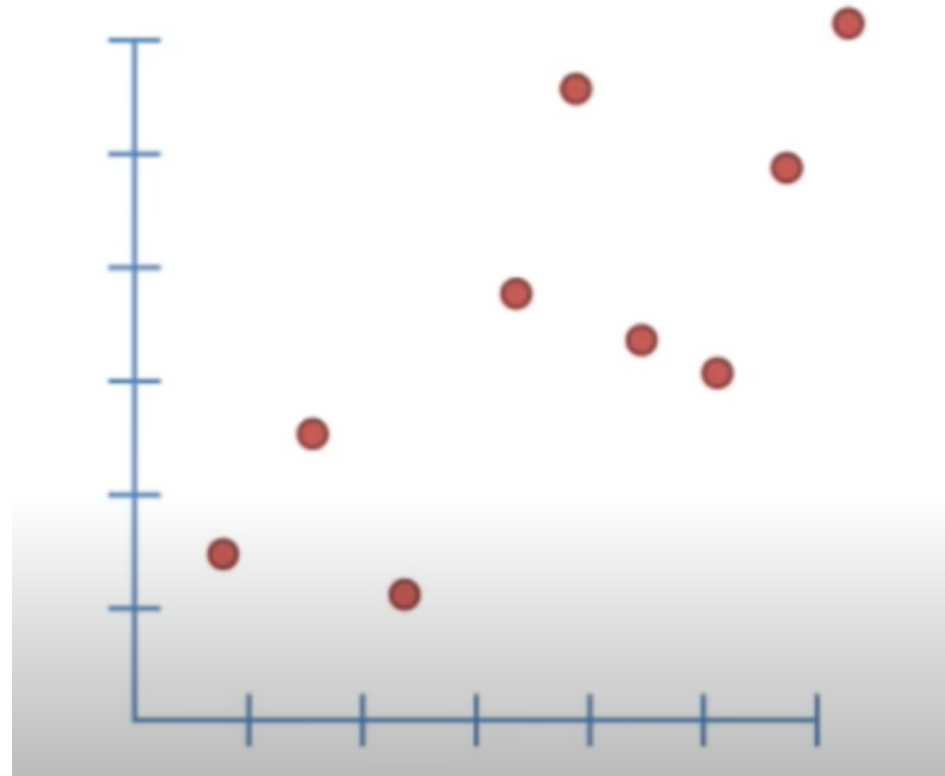
easy to eyeball (and to calculate) when just a few data points

parameters:
intercept = .943
slope = .622

($Y = .622X + .942$)



But what about when there are more data points?

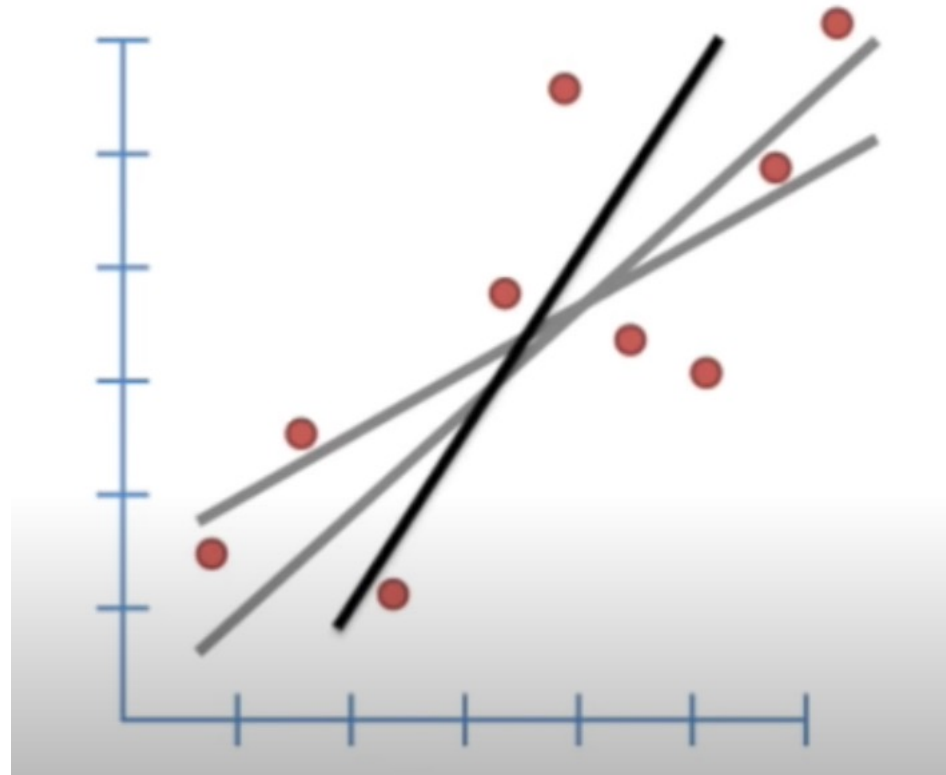


????

??

How do we know which line best fits the relationship?

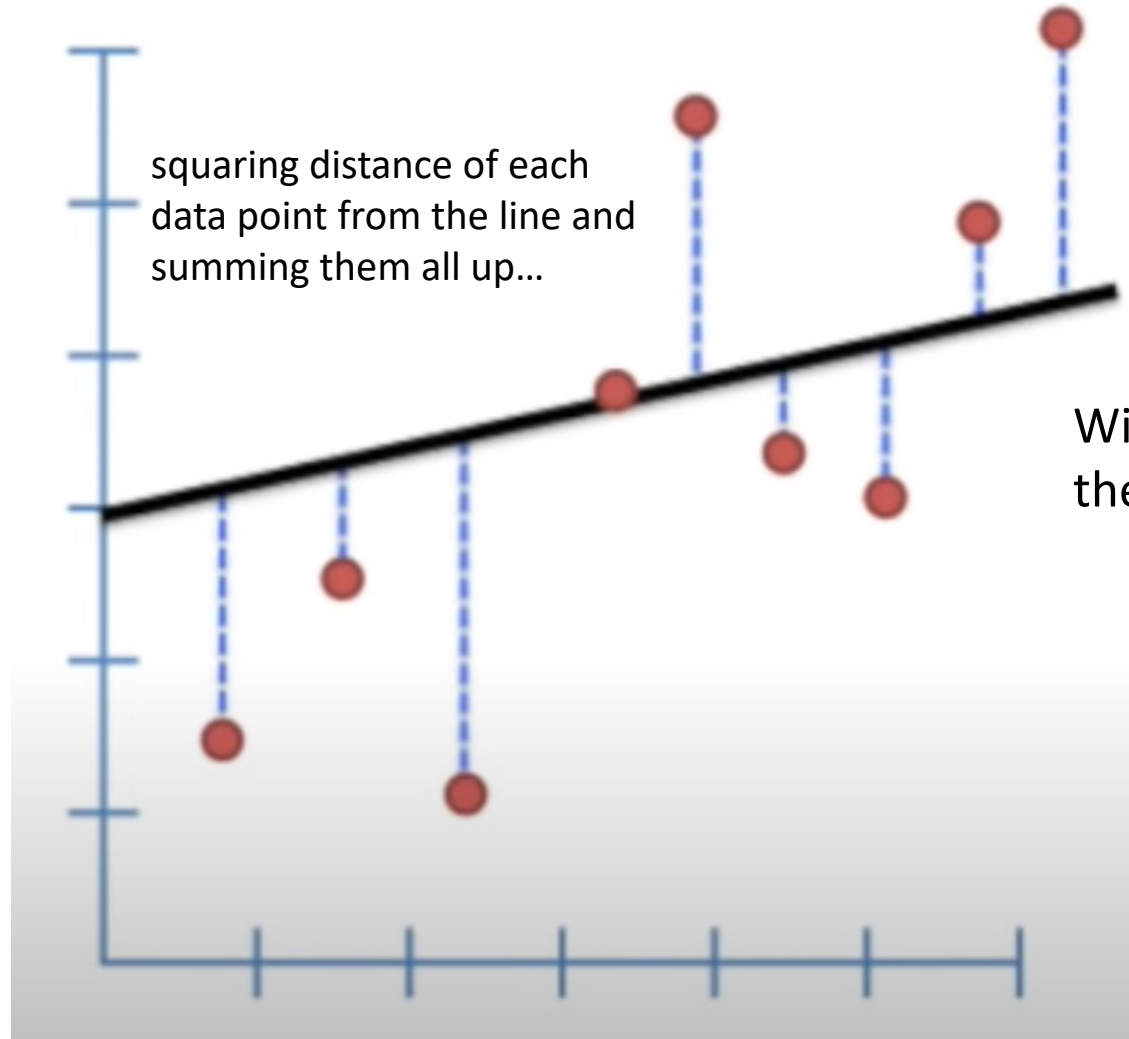
And how does the computer figure it out?

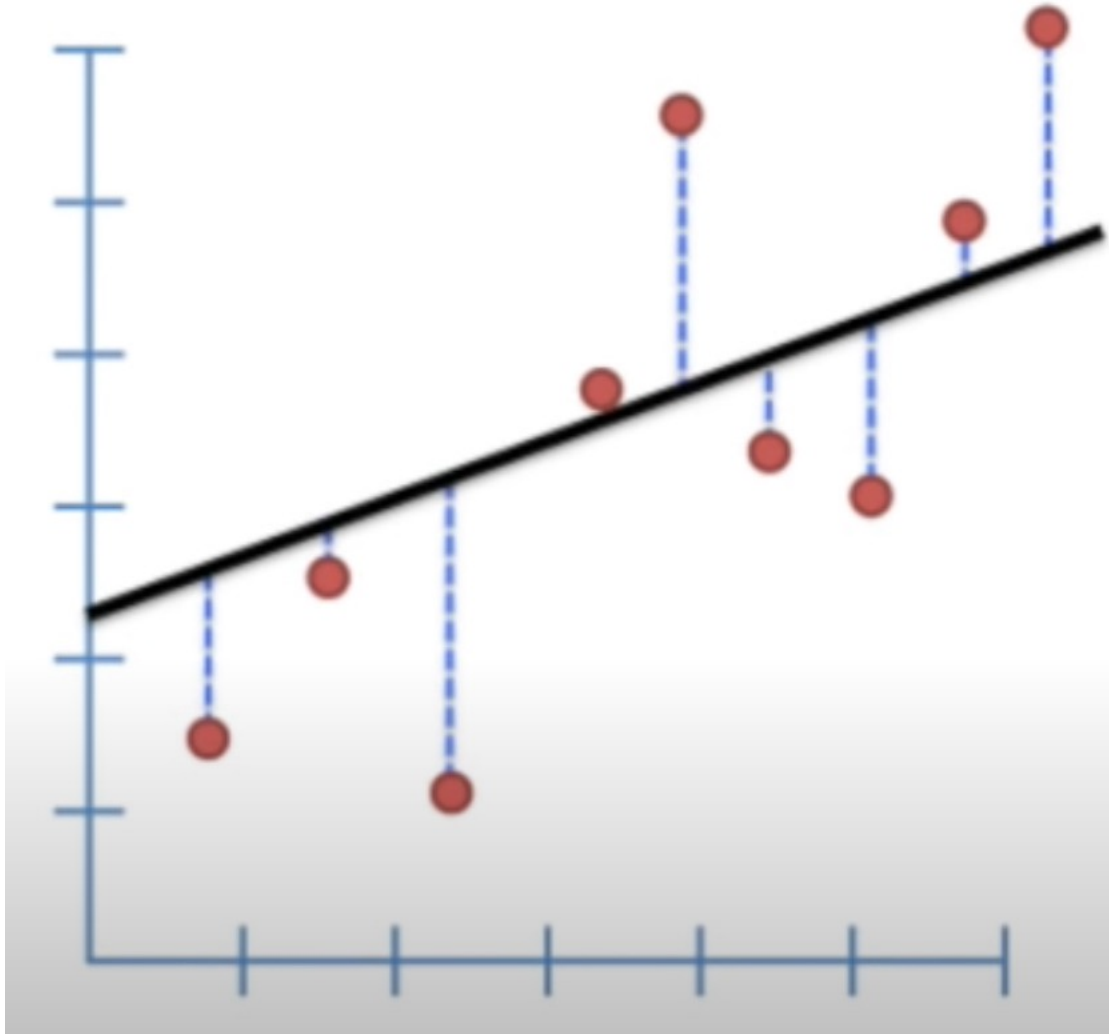


????

??

The answer lies in calculating **sum of squared residuals**





With this line,
(different slope and intercept)
the sum of squared residuals = 14.05

lower is better!

we want to find the line that
gives us the **least sum of squared
residuals**.

This is called
minimizing the loss function

There is an exact **analytic solution**...



$$X^T X = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ x_1 & x_2 & x_3 & \dots & x_n \end{bmatrix} x \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \cdot & \cdot \\ \cdot & \cdot \\ 1 & x_n \end{bmatrix} = \begin{bmatrix} N & \sum X_i \\ \sum X_i & \sum X_i^2 \end{bmatrix}$$

$$X^T \vec{y} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ x_1 & x_2 & x_3 & \dots & x_n \end{bmatrix} x \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ y_n \end{bmatrix} = \begin{bmatrix} \sum y \\ \sum X_i y_i \end{bmatrix}$$

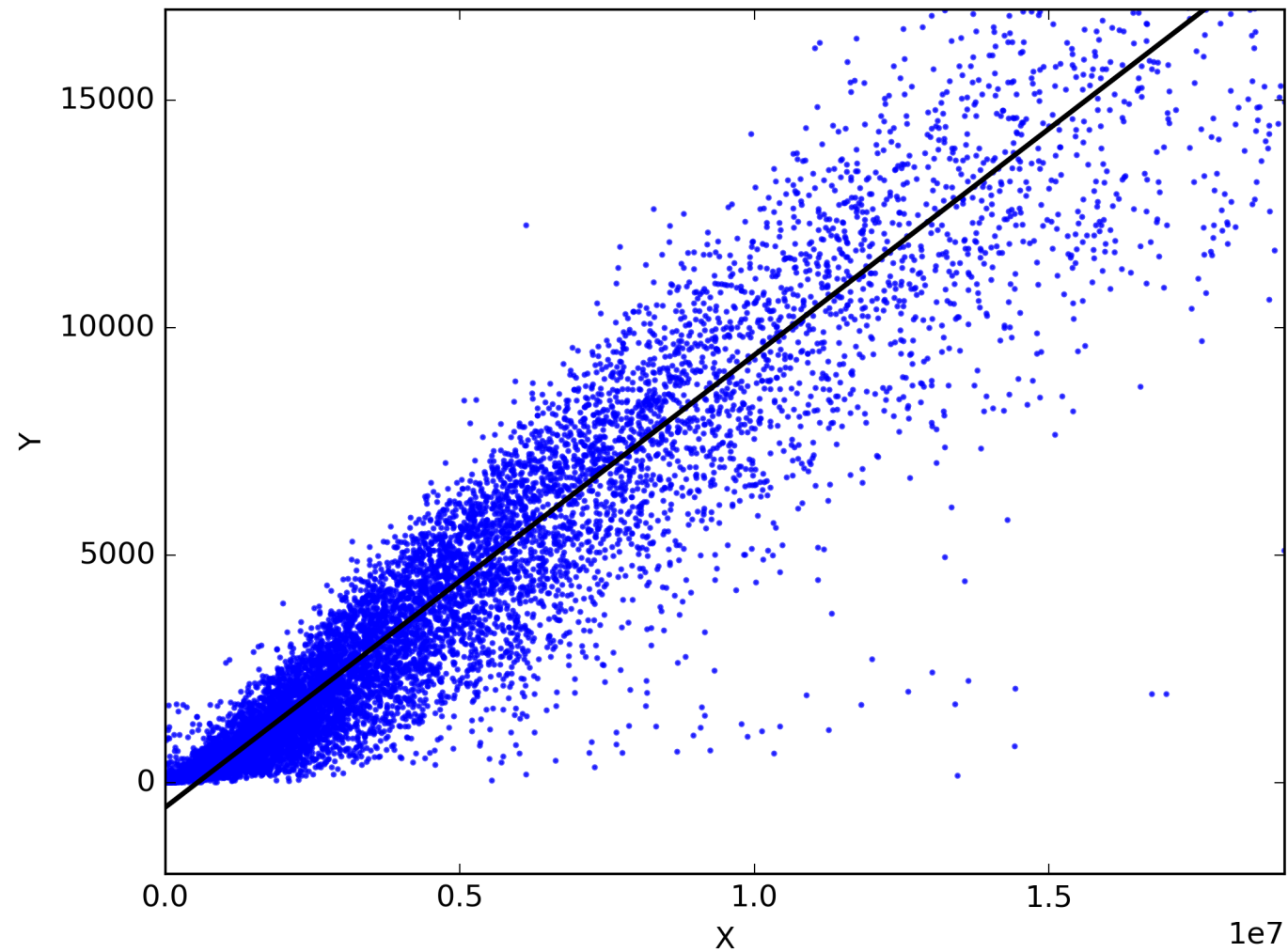


$$\vec{\beta} = (X^T X)^{-1} X^T \vec{y}$$
$$\begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = \begin{bmatrix} N & \sum X_i \\ \sum X_i & \sum x_i^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum y \\ \sum X_i y_i \end{bmatrix}$$

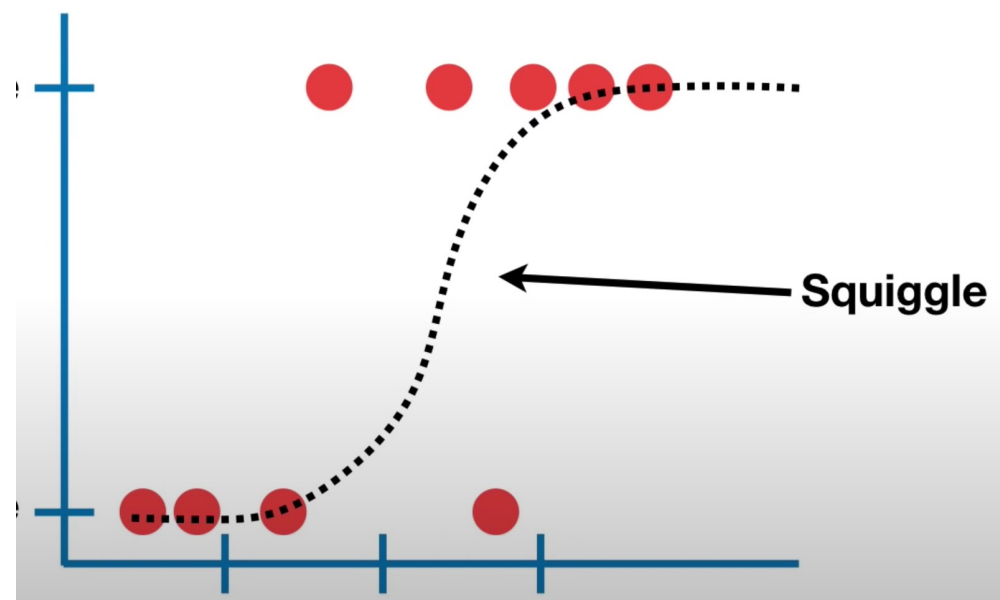
But it's complicated even for the computer and takes a ton of computing resources!

Plus, it only works for simple linear regressions

it's very slow and computationally expensive
when there are lots of data points



And can't be applied directly to other types of regression
like logistic regression (can't identify squiggle of best fit!)



(process of repeated reviewing and testing)

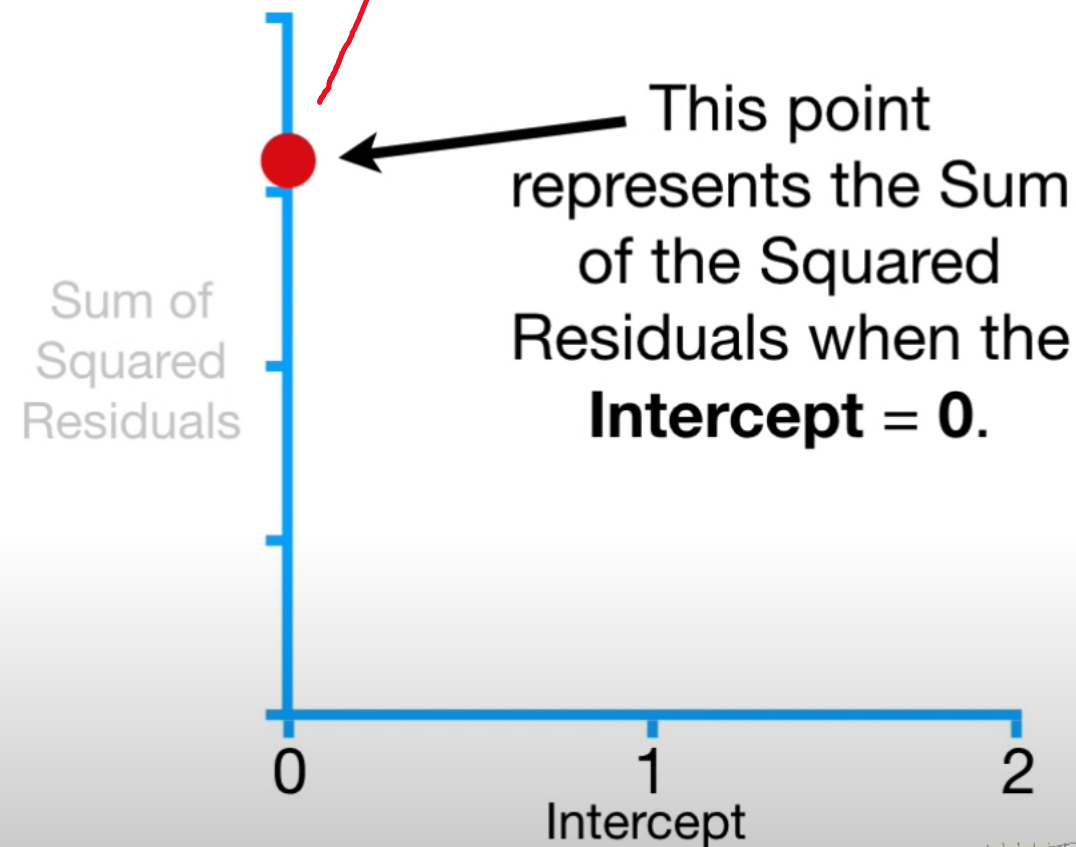
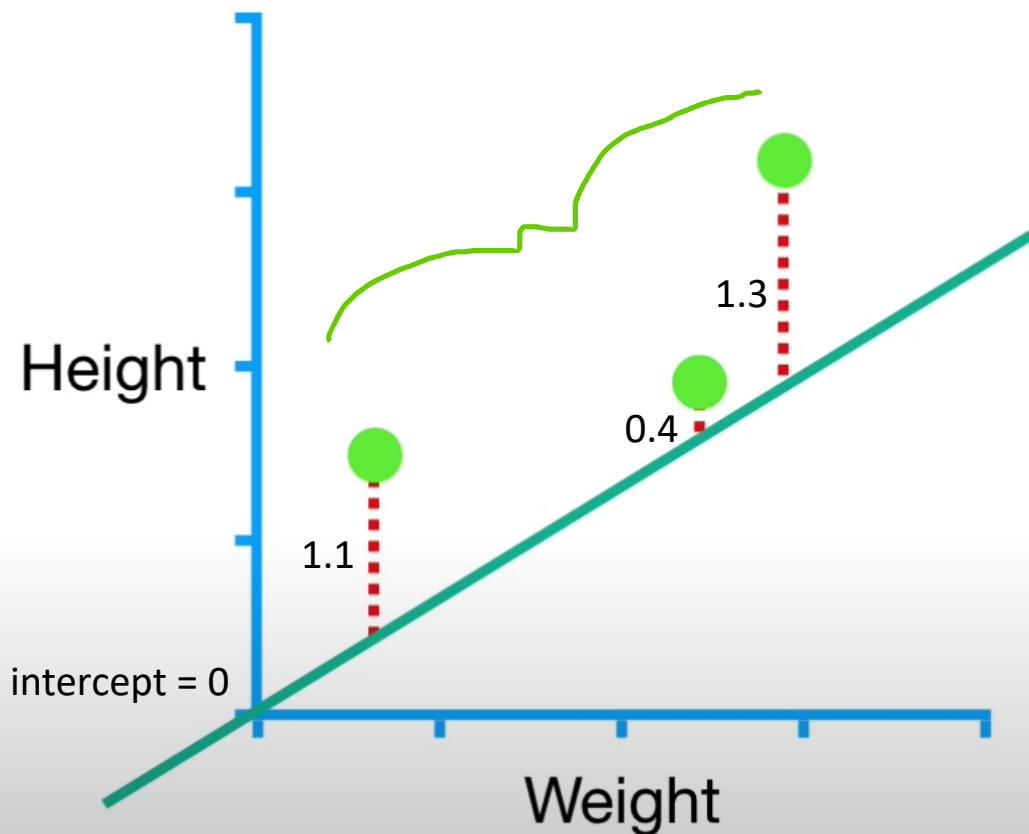
Luckily, there's an **iterative solution** that can estimate the optimal parameters, while using way less computer resources, and work in way more situations...

Gradient Descent



Let's start with 3 datapoints and a line with intercept 0

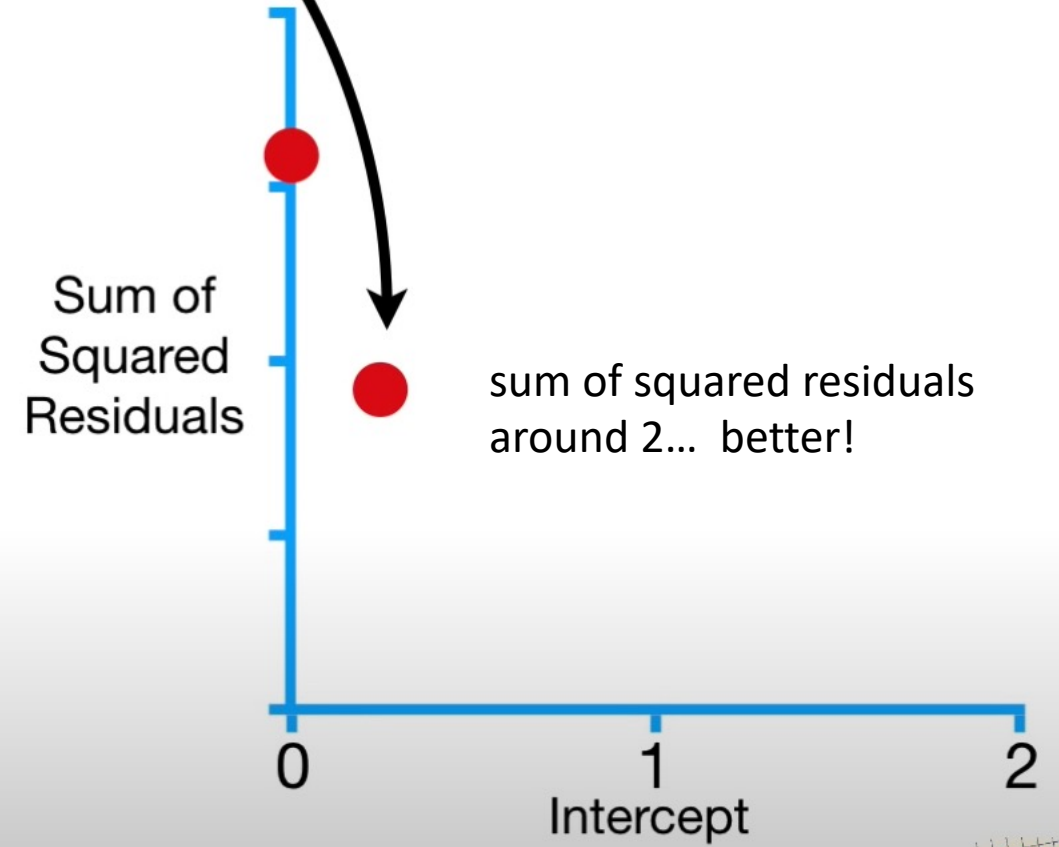
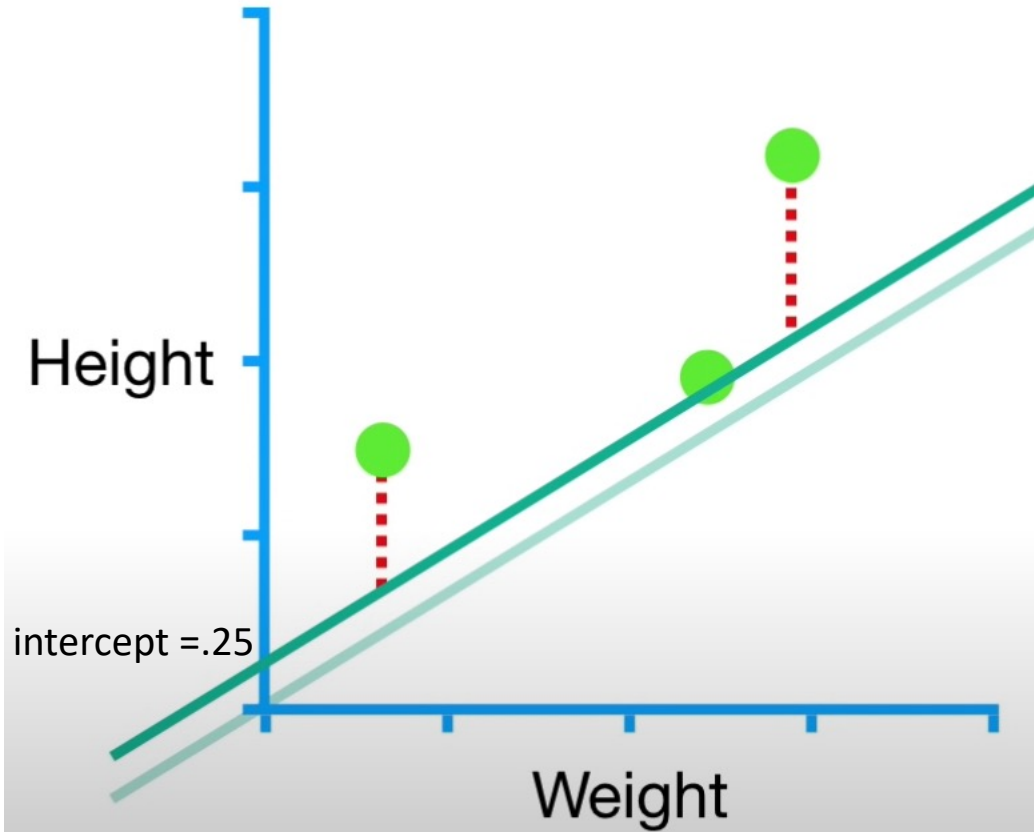
$$\text{Sum of squared residuals} = 1.1^2 + 0.4^2 + 1.3^2 = 3.1$$



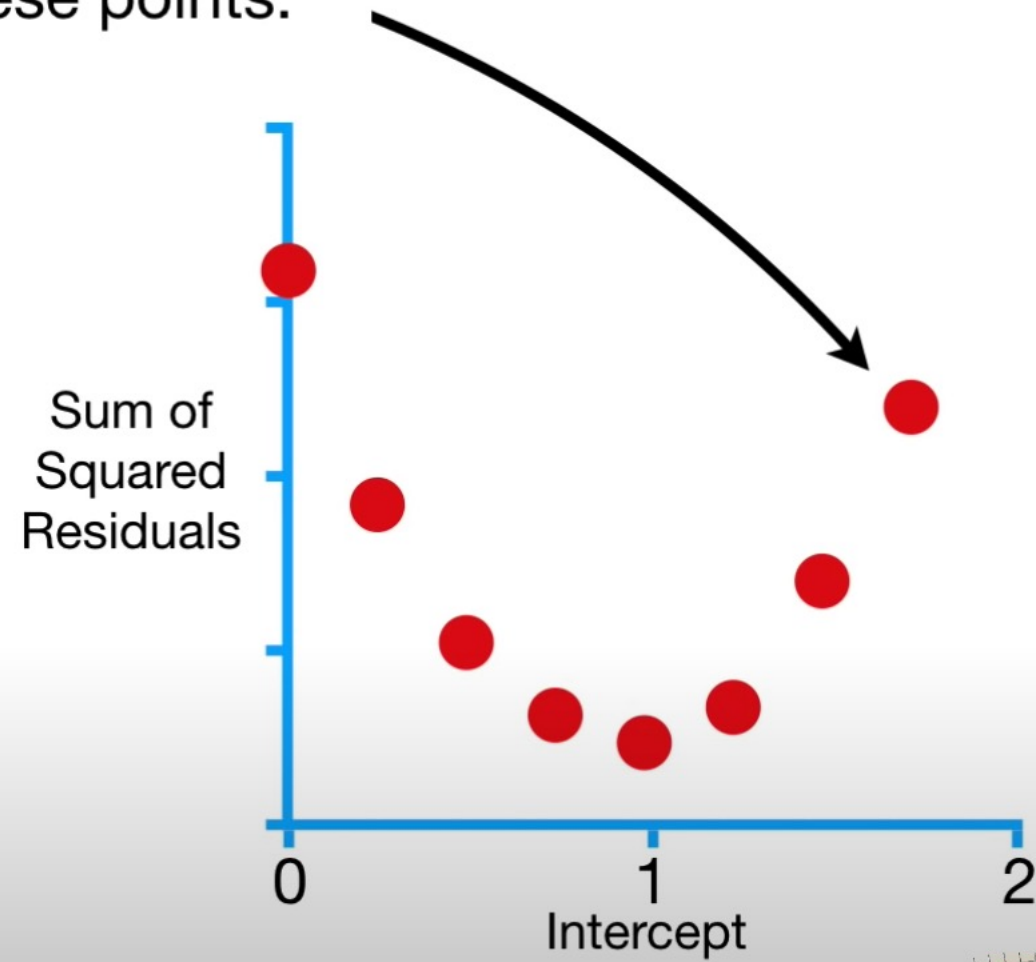
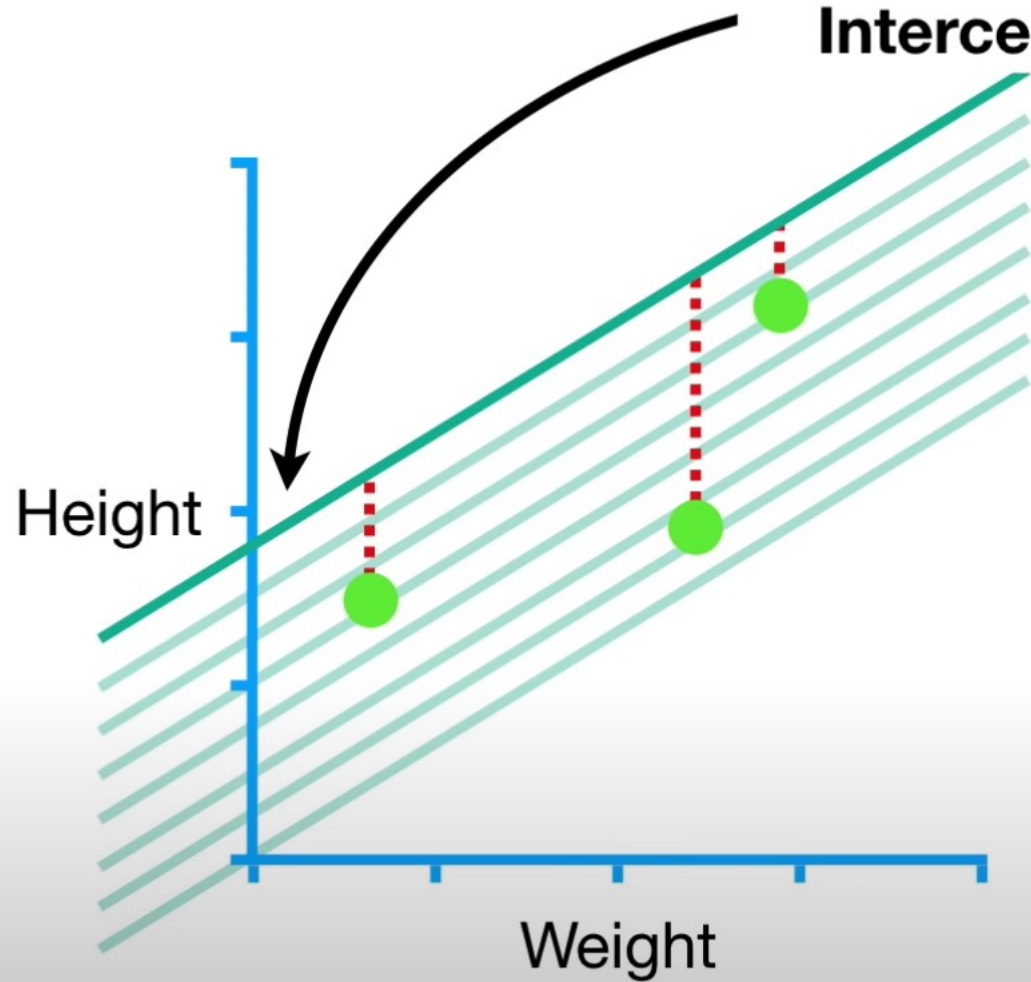
now let's change the intercept to .25



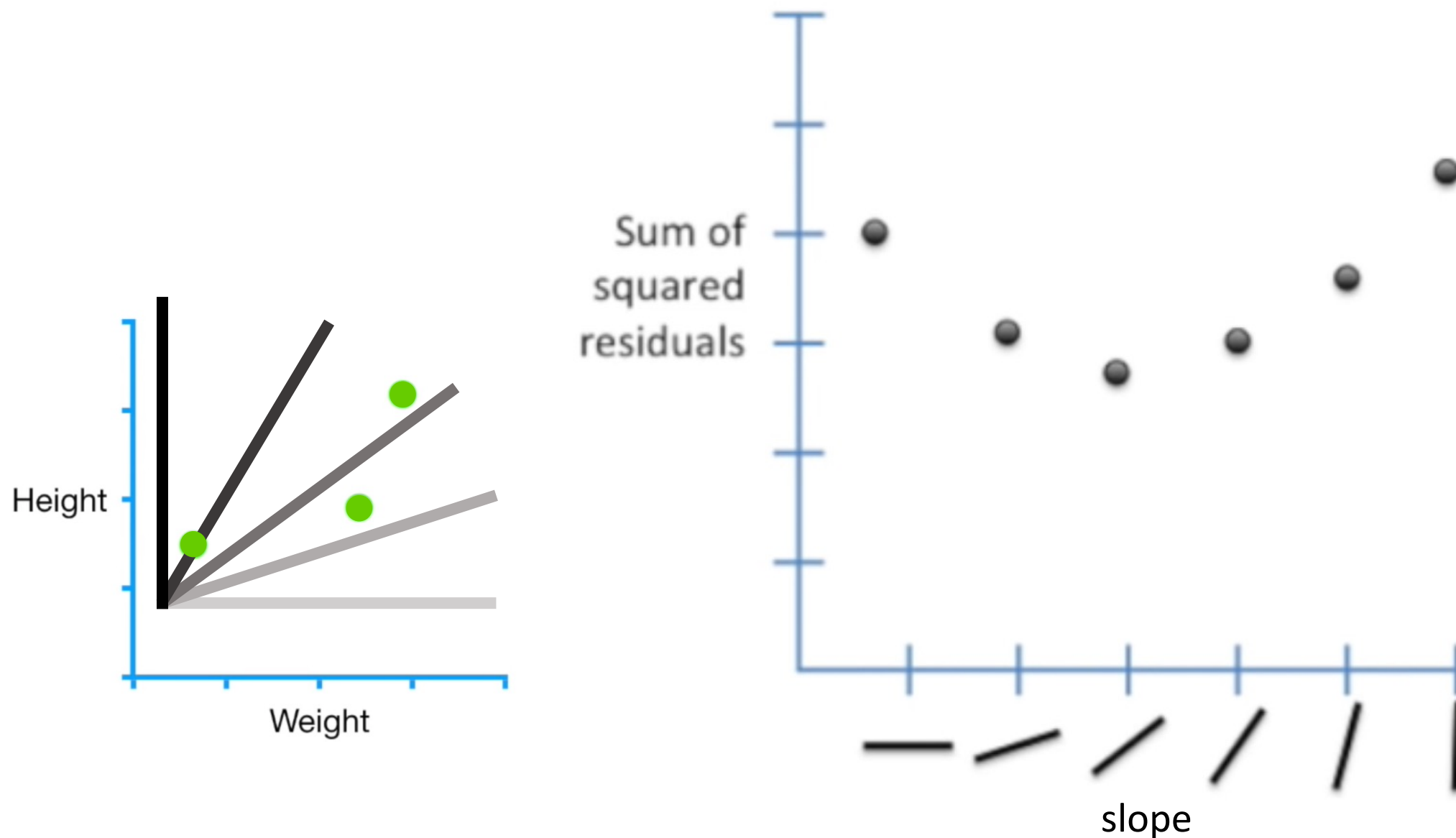
...then we would get
this point on the
graph.



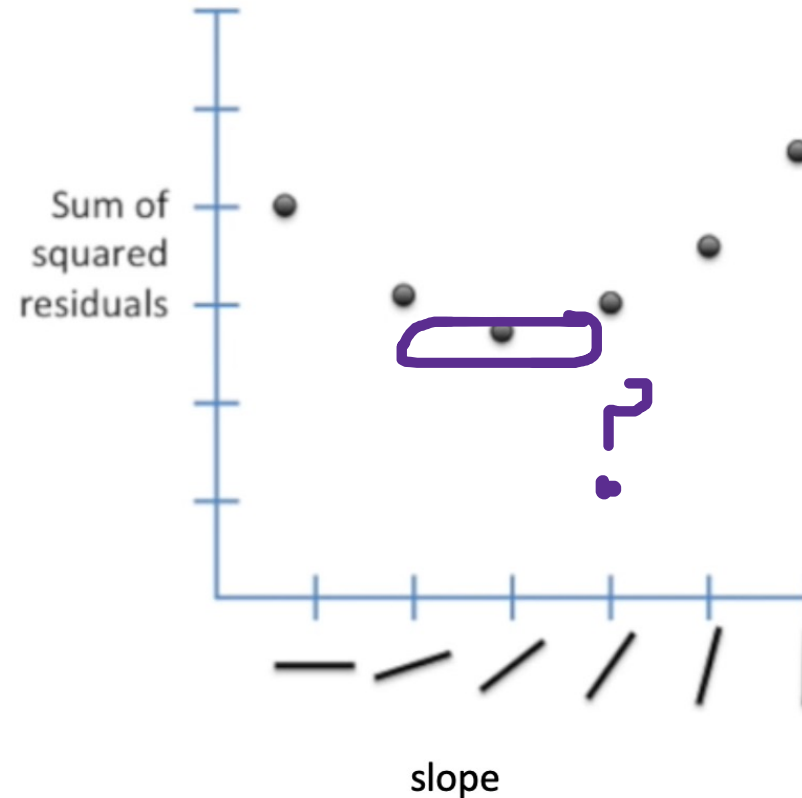
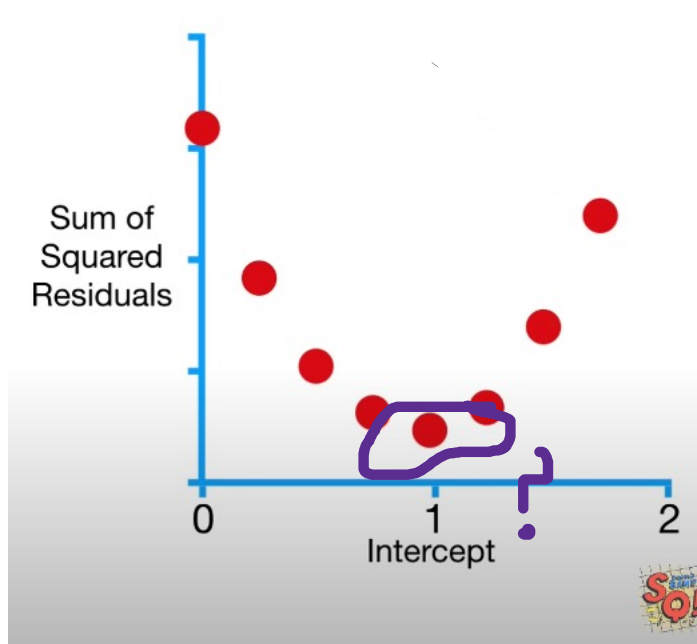
And for increasing values for the **Intercept**, we get these points.



Starting with a slope of 0, adjusting the slope bit by bit and plotting the Sum of Squared Residuals for each angle we try, we see a similar pattern...



Remember, we are looking for the Least Sum of Squared Residuals, so the optimal intercept and slope values must be at the bottom of each curve.



But in both cases, how do we know that we found the bottom of the curve?
What if the values we chose somewhat randomly missed the absolute bottom?

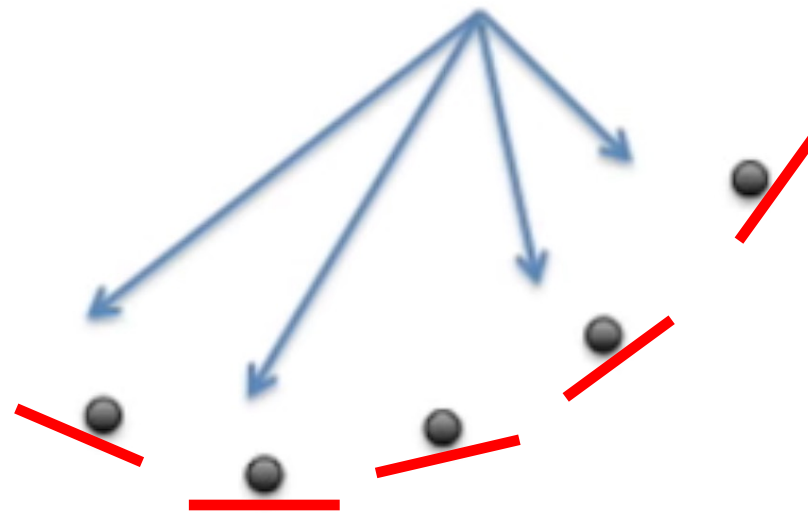
Gradient Descent is smart! It doesn't just guess random numbers hoping to find the least sum of squares by chance.

It adjusts its guess on each iteration based on...

The Derivative

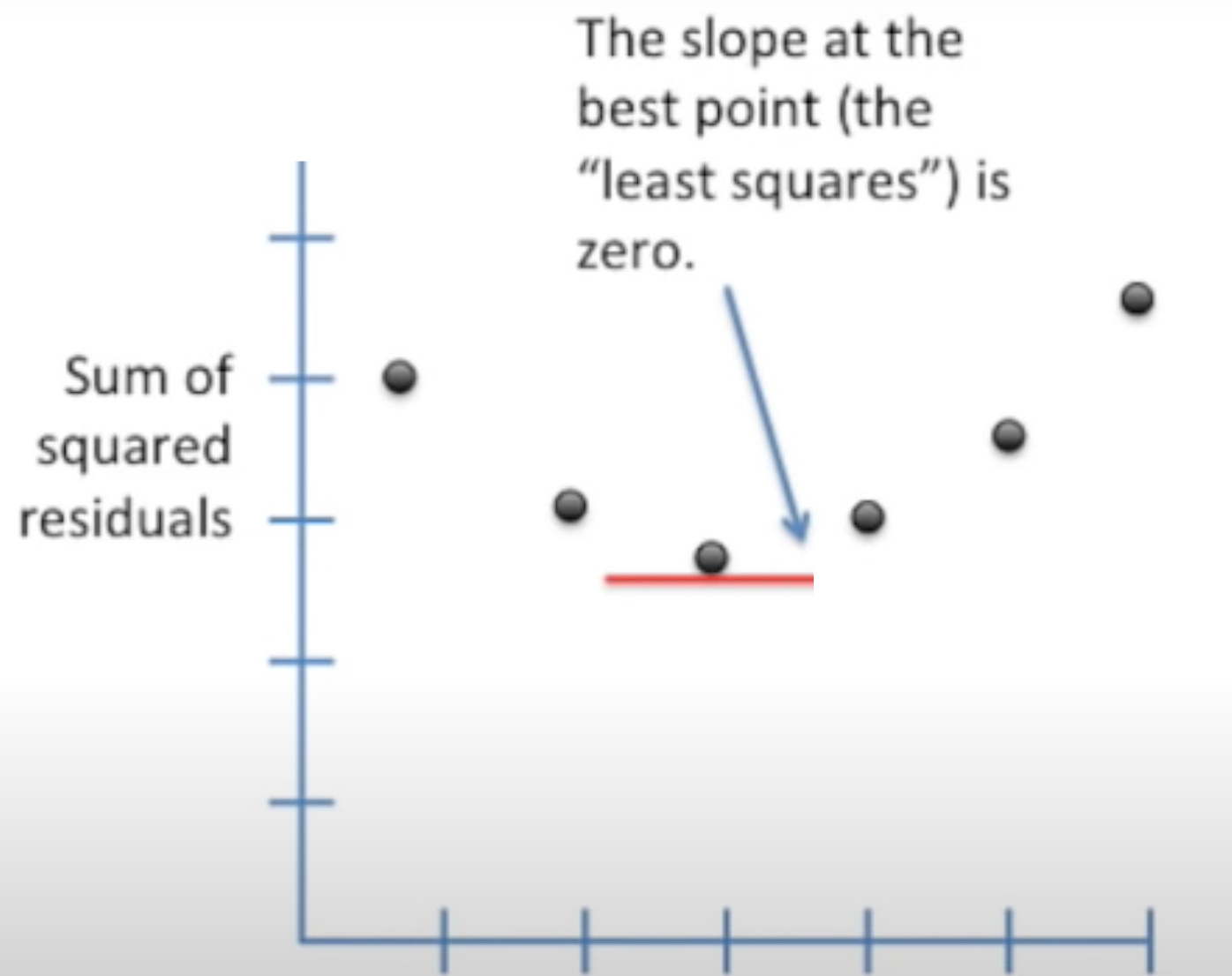


Sum of
squared
residuals



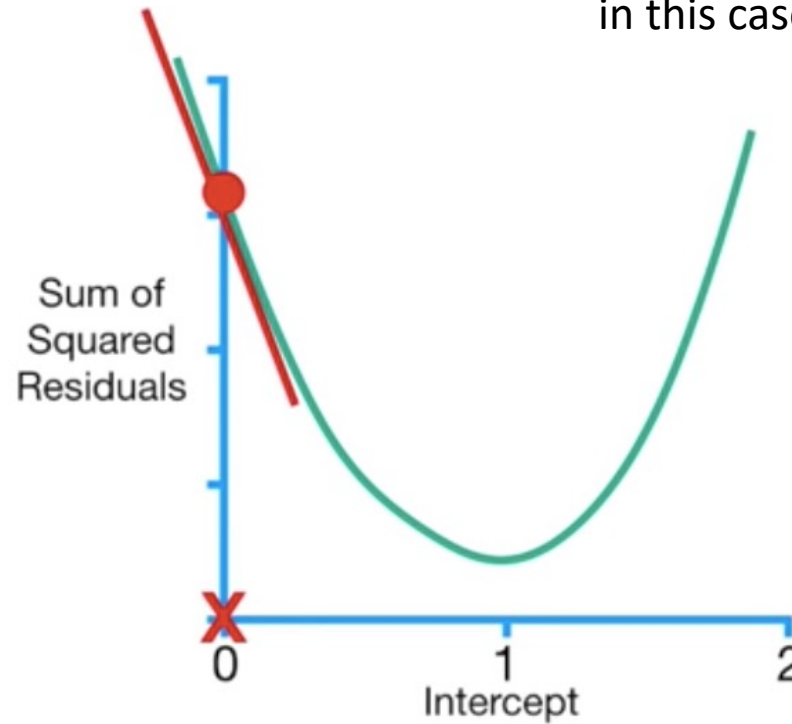
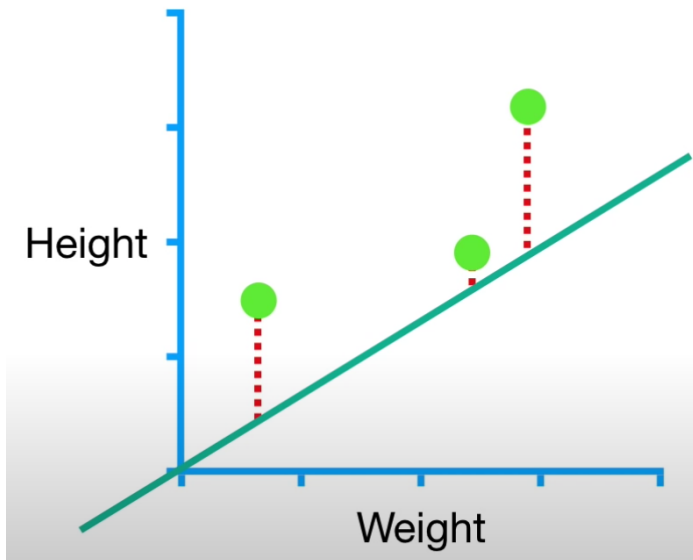
The derivative tells us the slope
of the function at every point.





One Dimensional Gradient Descent

(looking for the optimal value for one parameter, in this case the intercept)



- choose random initial value for the parameter, (let's say intercept = 0 for this example) and plot sum of squared residuals at that point

- calculate **derivative** (slope of the function) at that point:

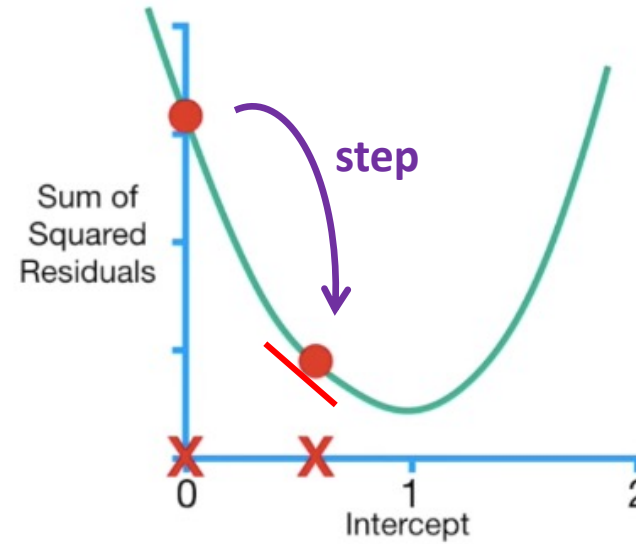
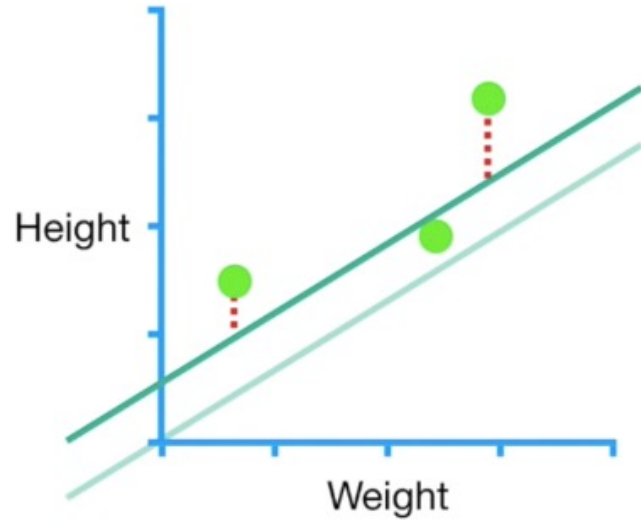
-5

- calculate **step size**: **derivative** * **learning rate** (.1)

-5 x .1 = -.5

hyperparameter

(we choose this number ourselves. higher learning rate = bigger steps, lower learning rate = smaller steps)

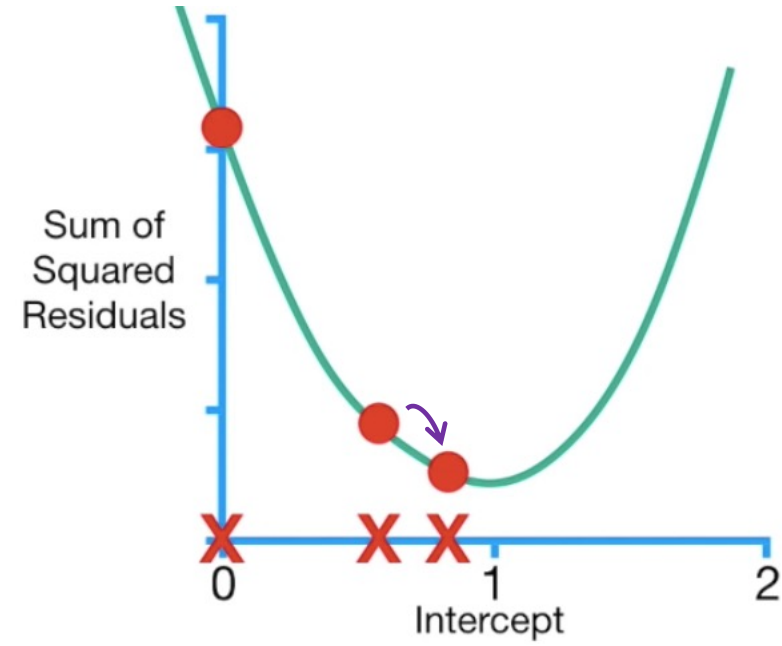
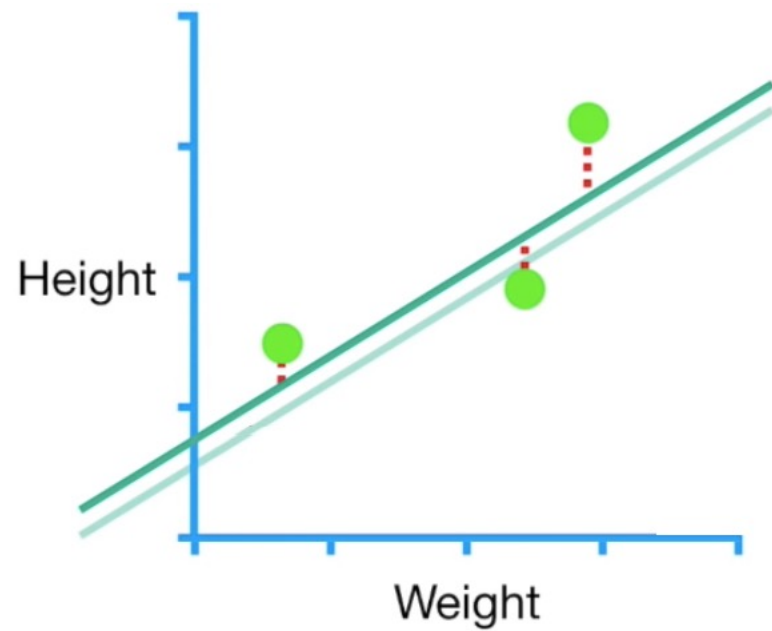


take a step!

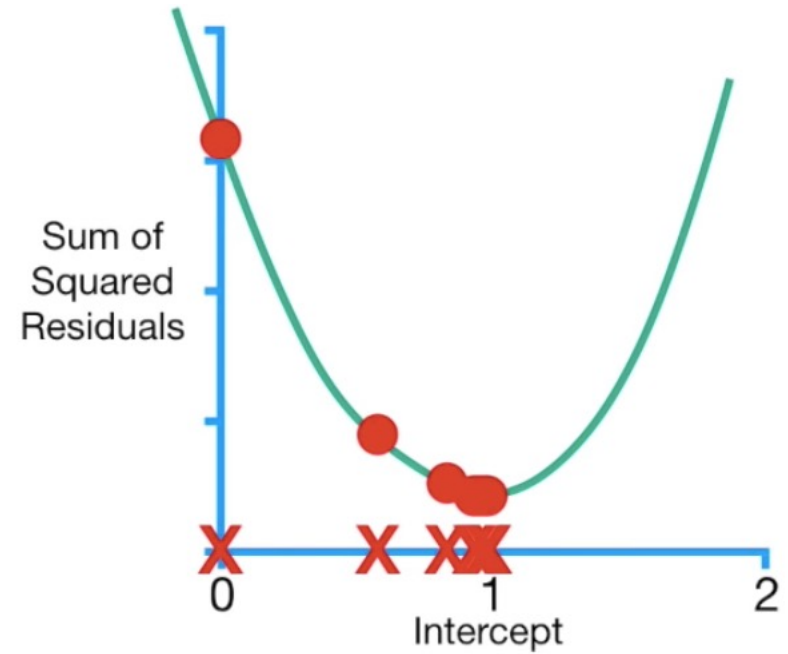
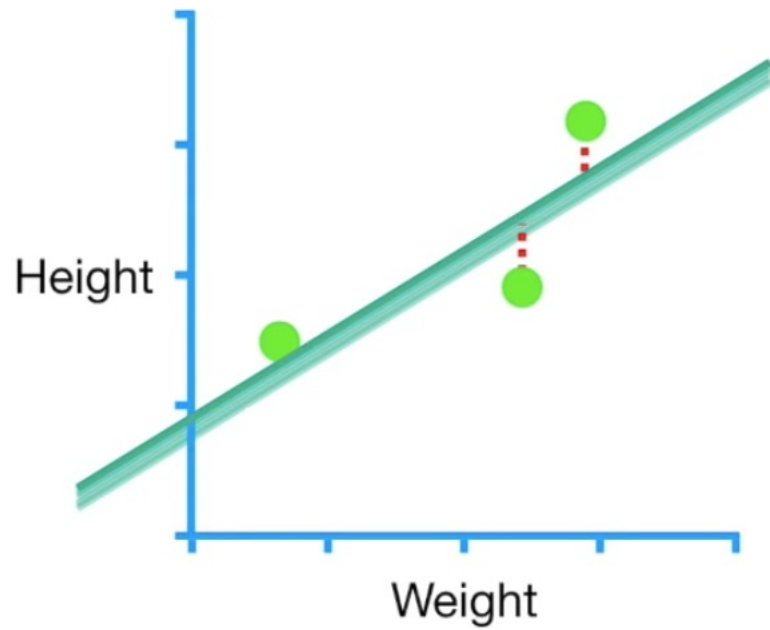
opposite direction of the **derivative**,
so here we move .5 (**step size**) to the right
new intercept = .5

We just did 1 **epoch** (iteration).

Now we repeat from our new intercept value .5:
calculate new **derivative** -> calculate new **step size** -> **take step**




As the **Loss** (sum of squared residuals) approaches its minimum, the **derivative** gets smaller, and so do the **steps**.



This makes Gradient Descent computationally efficient.
It does a few calculations far away from the minimum,
and more calculations as it approaches the minimum of the Loss Function.

When does it stop?

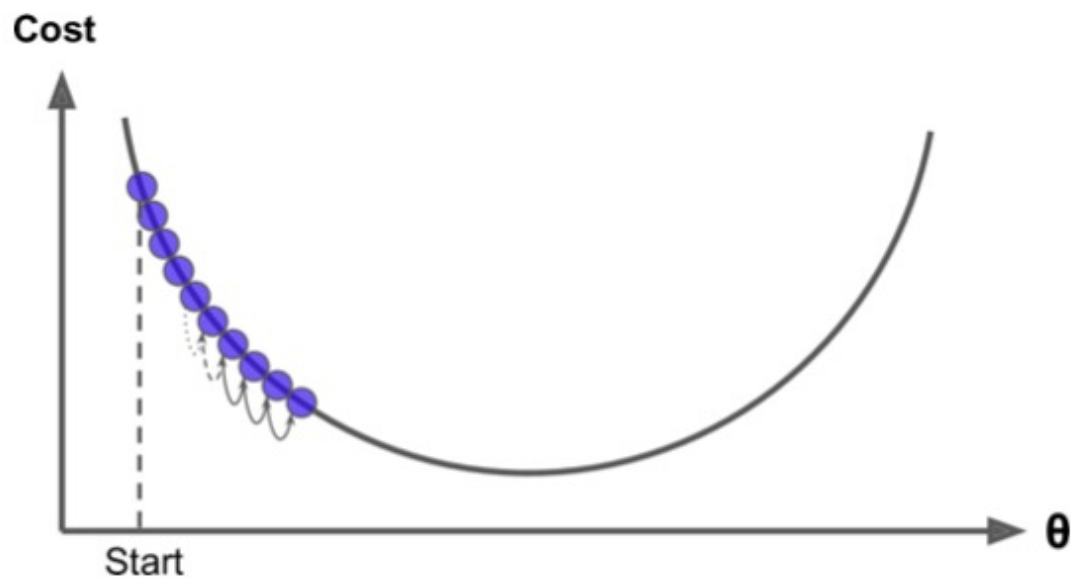
 hyperparameter
(we get to choose!)

The Gradient Descent algorithm can have different **stopping criteria** :

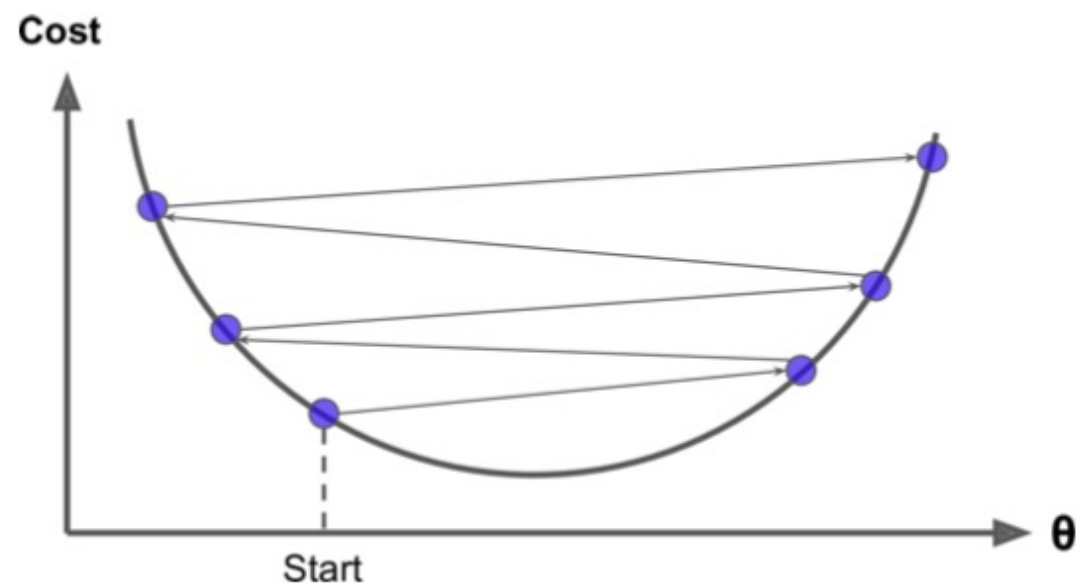
- **Minimum Step Size** (e.g. 0.001). When the step size is smaller, the Gradient Descent has converged, and the corresponding intercept is the optimal value.
- **Maximum Number of steps** (e.g. 1000)

We must be careful choosing the learning rate, as it affects the step sizes.
too small or too large steps and gradient descent won't find the minimum of the loss function!

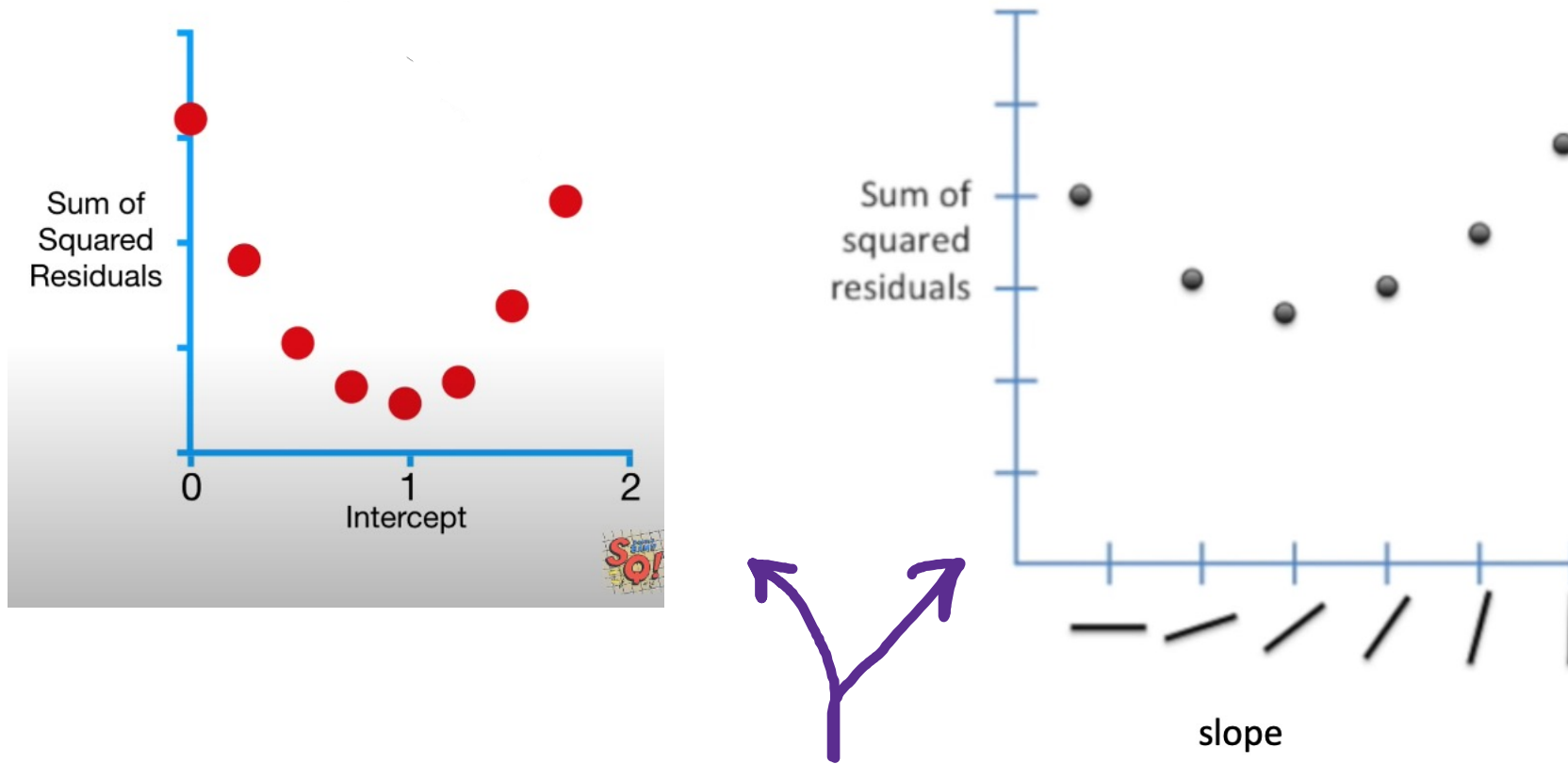
Learning rate too small



Learning rate too large



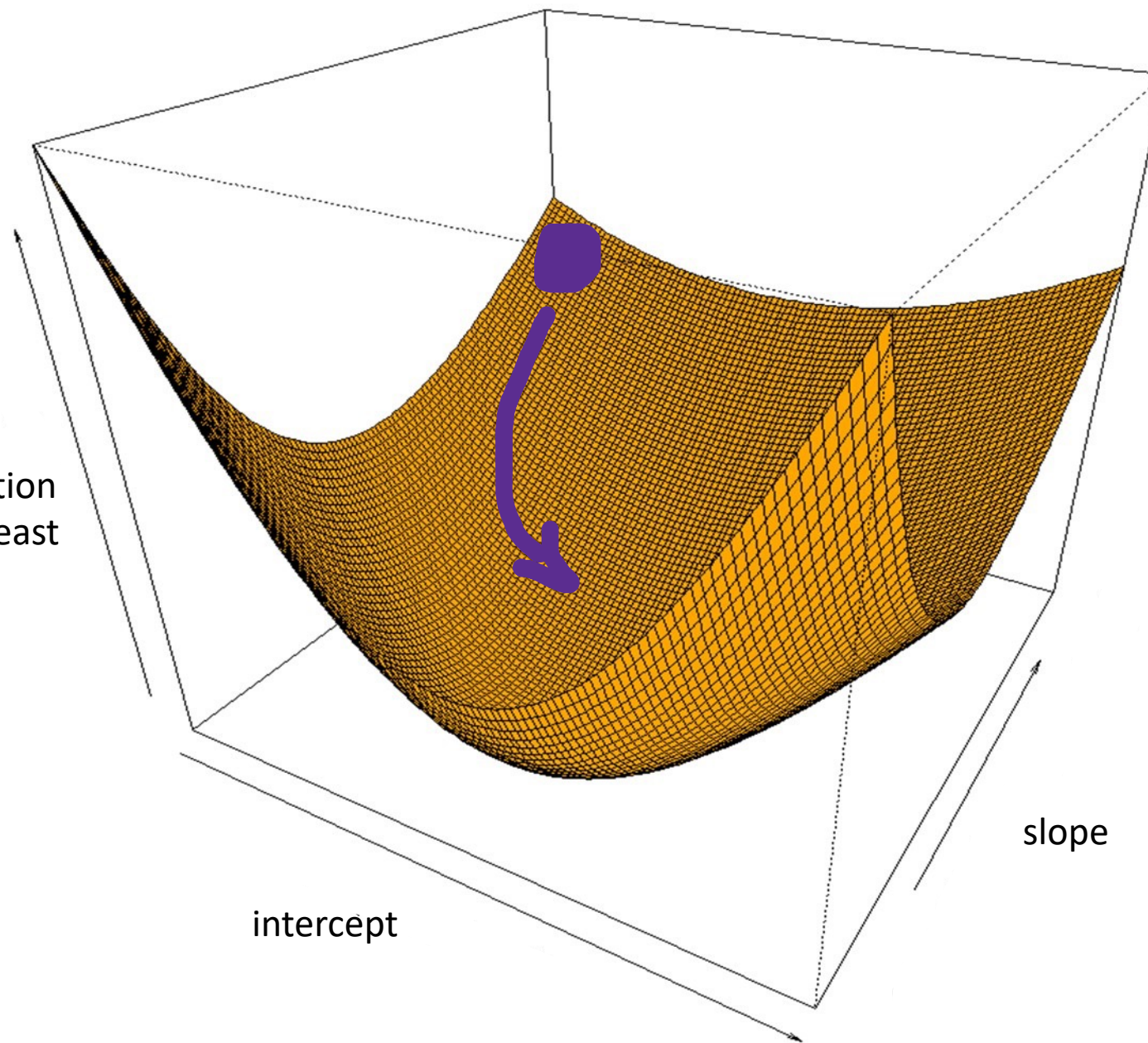
We've been talking on one-dimensional gradient descent, looking for the optimal value for one parameter at a time...



But in reality, gradient descent looks for the optimal value for intercept and slope simultaneously!

This is called the
“**energy landscape**
of loss function”

loss function
(sum of least
squares)



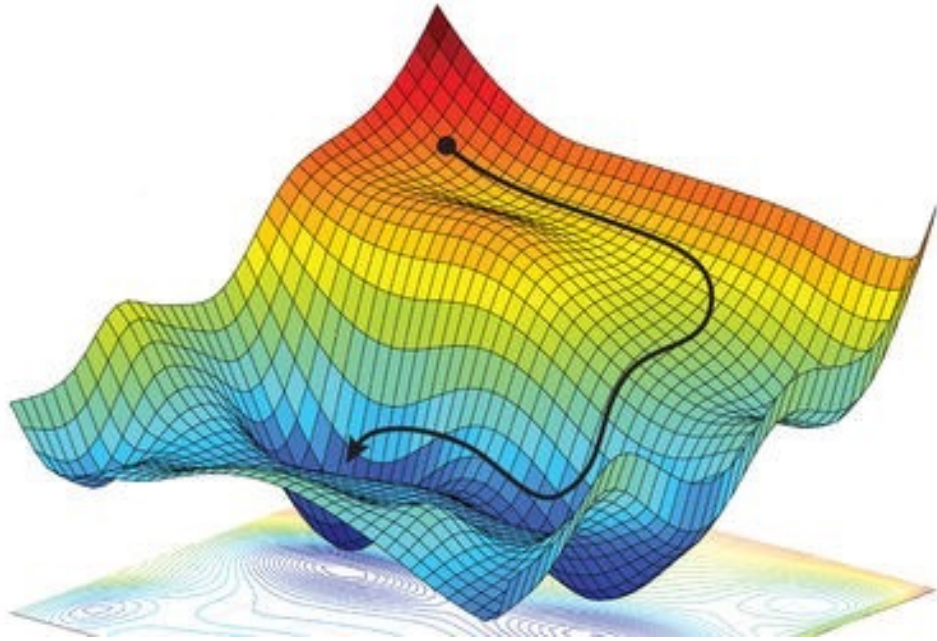
starting at a randomly
chosen **slope** and
intercept...

calculating derivatives
of **both** at each epoch...

taking a step in the
direction and step size
set by the derivatives...

and repeat.

kind of like a ball rolling
down the energy
landscape until it slows
down and stops
at the bottom
(**minimized loss
function**)

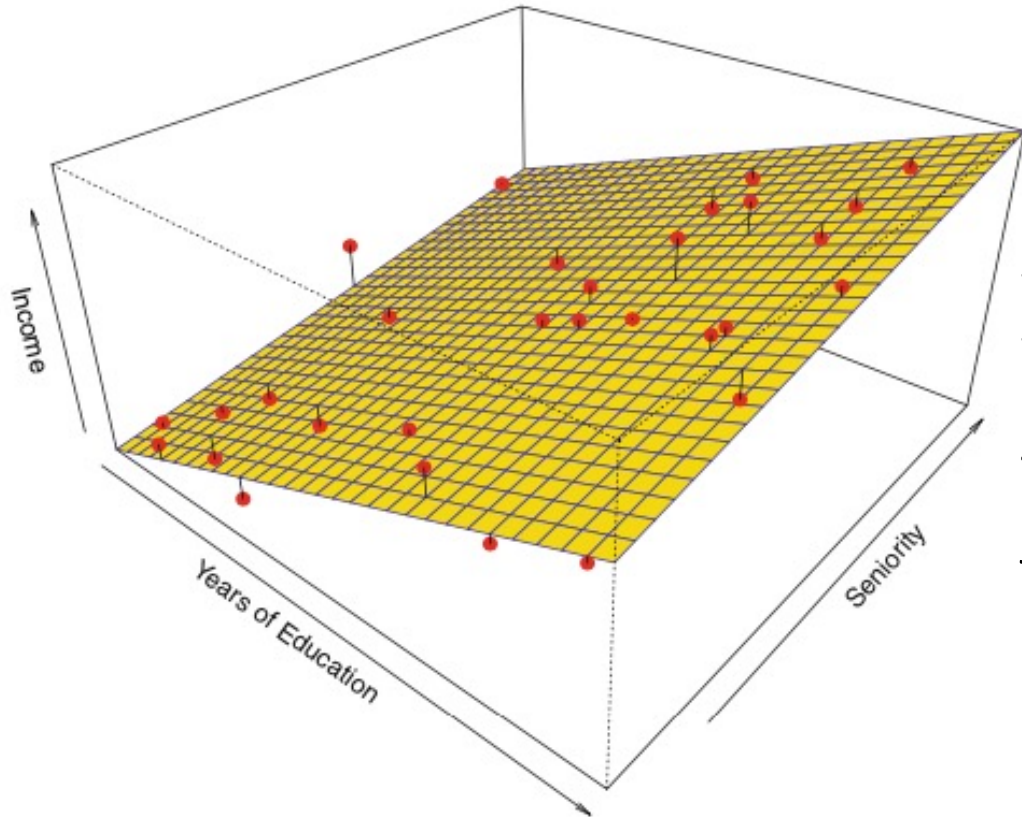


The energy landscape of the function isn't always so smooth and minimal loss isn't always so easy to find!

Some models/loss functions are much more complicated than simple linear regression/sum of squared residuals. They can have very complicated energy landscapes with 'pockets' the gradient descent could get 'stuck' in, preventing it from reaching the global (true) minimum.

We can try avoid this by adjusting the **learning rate** and **stopping criteria** hyperparameters.

what about a problem with 3 features?



then we're looking for a **plane of best fit** (instead of line of best fit) to minimize the loss function.

The loss function will still be sum of squared residuals, just calculated using the data points' distances from the plane instead of from a line.

The associated energy landscape of the loss function is in **4D**. We can't visualize it, but gradient descent still works the same to find its global minimum, calculating derivatives and taking steps accordingly.



4, 5, 10, 100, 70000 features...
gradient descent will still work!

Thank You!

