# Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Spring 2023

# Contents

CHAPTER 3

# Gradient Descent

## What is gradient descent?

### 3.0.1 Why do we need gradient descent?

In the last chapter, we used an **analytical** approach to solve the OLS and RR problems.

By "analytical", we mean we got an **explicit** answer: an equation we can use to directly compute the correct answer.

The trouble is, we can't always do this:

- Sometimes the problem or the loss function can't be **rearranged** into a simple **equation**.
- Or, we have **too much** data, and directly computing the answer would take way **too long**.

**Concept 1**

Most **problems** we come across cannot be solved **analytically**.

Well, if we can't **directly** find the **best** answer, what's the next best thing? Finding a **better** solution than your current one.

So, our mission is to gradually try to find a better and better answer. This type of approach has a couple benefits:

- It's **quicker** to see if we're using a good model: if we're making very little progress, we can **quit** early and try something else.
- If we don't need **all** of our data to get the answer, we don't need to spend as much time. If our answer is **good** and not getting better, we can **stop**.
- It's easier to find a **better** answer than the **best** answer: our equations will be **simpler**. In some case, it might not have even been **possible** without this gradual approach!

**Concept 2**

When we can't reasonably find a **best** answer, it's often easier to find a **better** answer and gradually **improve**.

**Gradient descent** follows this philosophy: we gradually **update** our solution to make it better and better.

### 3.0.2 How do we improve?

So, now, the question is: how do we **improve** our hypothesis? We'll be modifying our hypothesis $\theta$ by some amount:

We'll do the same for $\theta_0$, but we'll do it separately. We'll come back to that.

$$\theta_{new} = \theta_{old} + \Delta\theta \tag{3.1}$$

**Notation 3**

In equations, we'll often use $\theta_{old}$ and $\theta_{new}$ to represent **before** and **after** we take a step.

We will use this notation **elsewhere** in the class.

So, we are interesting in $\Delta\theta$: how do we plan to change $\theta$? What does $\Delta\theta$ look like?

Well, we want to modify

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \tag{3.2}$$

So, we want to modify each of those terms.

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} + \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \\ \vdots \\ \Delta\theta_d \end{bmatrix} \tag{3.3}$$

So, we have our total change!

$$\Delta\theta = \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \\ \vdots \\ \Delta\theta_d \end{bmatrix} \tag{3.4}$$

Notice that the shape of this change matches the shape of $\theta$: $(d \times 1)$.

**Concept 4**

We need a **separate** term $\Delta\theta_i$ for each $\theta_i$ we want to **improve**.

So, a vector of the **total** change, $\Delta\theta$, needs to have the **same shape** as $\theta$: $(d \times 1)$.

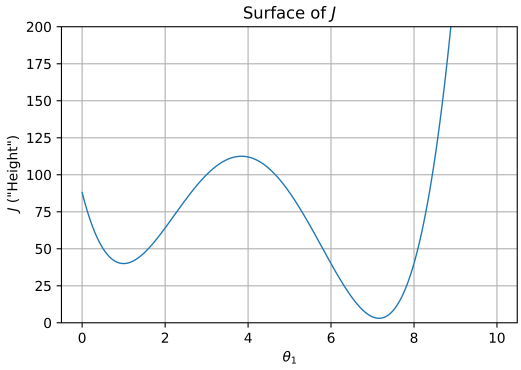
### 3.0.3 The name: "gradient descent"

Our goal is to gradually **decrease** J, step-by-step. We do this using the **gradient**, hence "gradient descent". Why the gradient? We'll discuss that later.

Reminder: Why are we "decreasing" J? Because J represents the "badness" of our model. We want it to be, simply put, "less bad".

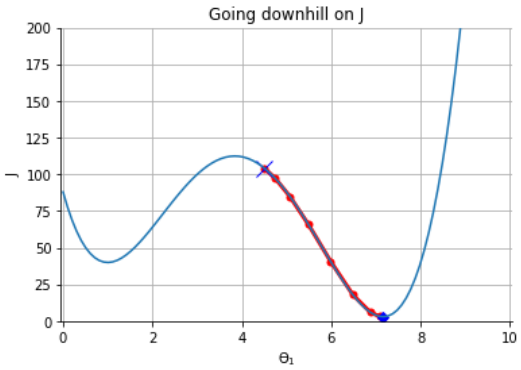
But why the word "**descent**"?

Our intuition is to imagine J as having a **height** at every input value. If you combine all of these different points, you get a **surface**, like the surface of a hill.




You can imagine this like some hills we want to "descend".

Then, decreasing J is moving **down** the the function, similar to rolling a ball down a hill. In other words, we **descend** the hill.




Like this! Starting from the blue "X", moving 'downhill'.

### 3.0.4 Input Space vs. Parameter Space

One more thing to note: we have two similar situations.

- $J$ is a **function** with $\theta$ as an **input**: $J(\theta)$.
- $h$ is a **function** with $x$ as an **input**: $h(x)$.

In both cases, we can imagine the **output** as the "**height**" of our function: the **hill** we mentioned before. This **physical** intuition is useful to **gradient descent**.

But, what about **input** to our function? That's the x-axis our hill is floating above:

- With $h(x)$, our x-axis was our **input space**, all possible $x_1$ values: the "space" containing all of our possible inputs.
- With $J(\theta)$, our x-axis is the **parameter space**, all possible $\theta$ values. We also called this our "**hypothesis space**".

We're assuming 1-D right now for simplicity. If we were 2-D, we'd have an entire 2D grid under our hill!

**Definition 5**

The **parameter space** is our set of all **possible** parameter combinations.

This is the same as the **hypothesis space**, because our parameters **define** our hypothesis.

When we **optimize** our hypothesis, we are "**exploring**" the hypothesis space.

- This can be seen as the "collection of **all possible models** in our model class".
- Why do we call it a "parameter **space**", not a "parameter set"?
  - It's the **structure**: the fact that some hypothesis are "closer" to each other: $\theta = 1$ is closer to $\theta = 2$ than $\theta = 10$

We mentioned one useful feature: we have a concept of which hypotheses are "**similar**": those which are **closer** in parameter space.

We've already used this fact! When normalizing towards the previous hypothesis, $\theta_{old} = 0$, we minimized $\|\theta - \theta_{old}\|$.

This is the **space** we're exploring, as we try to move **downhill**.

**Clarification 6**

Pay attention to your **axes**!

Sometimes, we're doing a 2-D or 3-D plot of J, and our inputs are $\theta_k$. Other times, we're plotting hypothesis $h$, with our input axes $x_i$.

These two plots could have the same surface, but they **represent** completely different things.

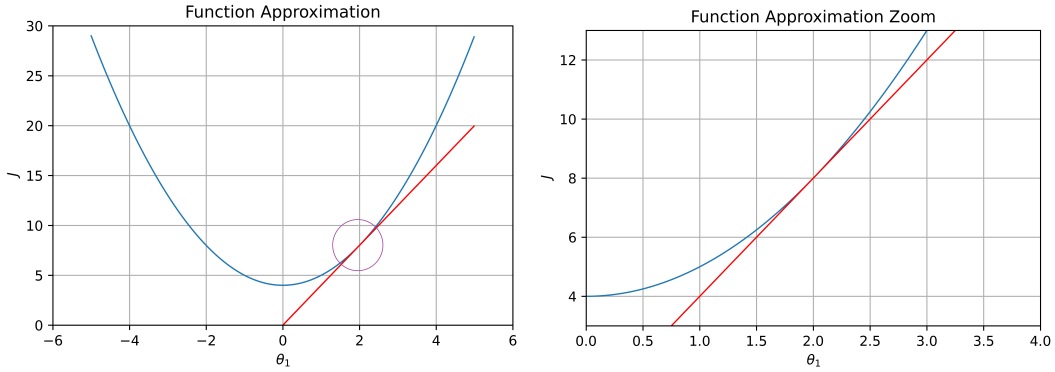
## 3.1 Gradient Descent in One Dimension

### 3.1.1 Derivatives (Review)

Here, we'll use some concepts from **calculus**.

We'll make improvements in small **steps**. And, we measure our improvement against the **loss function**, J: that's what we want to **optimize**.

In calculus, we found that, over **small** enough steps, you can **approximate** a smooth function as a straight line.




It looks more like a line as we zoom in: hence the **local** approximation.

**Concept 7**

A **smooth** (enough) function can be **approximated** with a **straight line** if you **zoom** in on it enough.

Looking at it this way is called a **local** view.

### 3.1.2 Optimize with Derivatives: 1-D

This gives us the **slope** of the function locally. Last chapter, we used $\frac{dJ}{d\theta} = 0$ to get our **minimum**.

But, let's not get too greedy - we want to **improve** our hypothesis, **not** immediately try to find the **best** one.

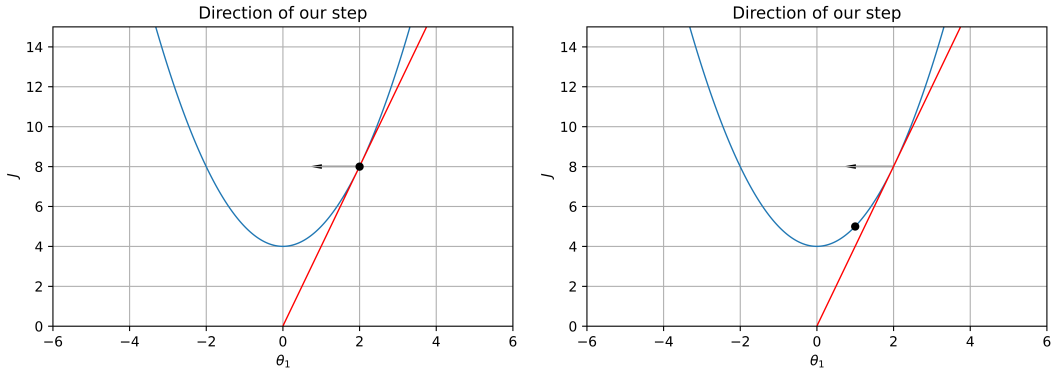
Because the best one might be expensive to find this way!

Well, what does our slope tell us? It tells us:

- How quickly J changes
- Whether it **increases** or decreases as we change $\theta$

That second one tells us *how to change* $\theta$: we want to move in the direction that **decreases** J.

If the slope is **positive**, then we want to **decrease** $\Delta\theta$: the sign of $\Delta\theta$ is the opposite of our desired change!




Our slope is **positive**. We want to **decrease** our function, so we move in the **negative** direction, and "fall down" the surface.

And so, for now, we have

$$\Delta\theta = -\frac{dJ}{d\theta} \tag{3.5}$$

**Concept 8**

In **1-D**, you can use the **derivative** to **optimize** our function J.

The **derivative** tells us how to immediately adjust $\theta_i$ to **improve** our J **locally**: we move in the **opposite direction**.

This gives us a procedure for optimizing J: get the derivative $J'(\theta)$, and repeatedly adjust $\theta$ in the opposite direction until you're satisfied.

We'll need to pick a condition for being satisfied, but we'll get to this later

There's a certain way this feels like we're moving "**downhill**": we're moving "down" the slope, to try to find a local **minimum**.

### 3.1.3 Convergence

If you do this procedure with the above equation, though, you'll often run into **problems**. Why is that?

Well, because each of your steps is too **big** or too **small**: we won't be able to find a **stable** answer, i.e. **converge**!

What does it mean to **converge**?

It means we get a **single answer** after repeated steps: given enough time, we'll get **close as we want** to one number, and **stay there**.

**Definition 9**

If a sequence **converges**, then our result gets as **close as we want** to a **single number**, without going **further away**.

**Example:** The numbers $1/n$: $\{1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots\}$ converges to 0.

If our answer **doesn't** converge, then it **diverges**. We can see why this might be bad: if we never **approach** a single answer, how do we know what value to **pick**?

### 3.1.4 Convergence: A little more formally (Optional)

Let's be more specific. Our sequence $S$ will converge to $r$.

$$S = \{s_1, s_2, s_3, s_4, \dots\} \tag{3.6}$$

"As close as we want": let's say we want the maximum distance to be $\epsilon$. That means, no matter what $\epsilon > 0$, we'll get closer at some point: $|m - s_i| < \epsilon$

$$|m - s_i| < \epsilon \text{ for some } i \tag{3.7}$$

"And stay there": at some time $k$, we never move further away again:

**Definition 10**

If a sequence $S$ **converges** to $m$, then for all $\epsilon > 0$, we can say

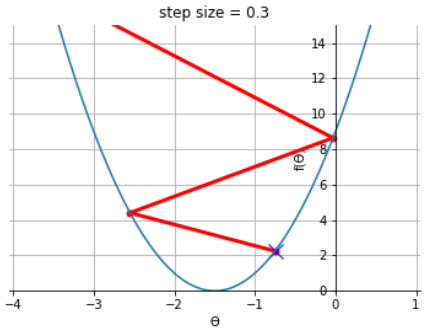
$$|m - s_i| < \epsilon \text{ for all } i > k \tag{3.8}$$

This is a "formal" definition of convergence.

### 3.1.5 Step size

If your steps are too **big**, your result might **diverge**: you make such big jumps, you move **away** from the minimum, and get worse.

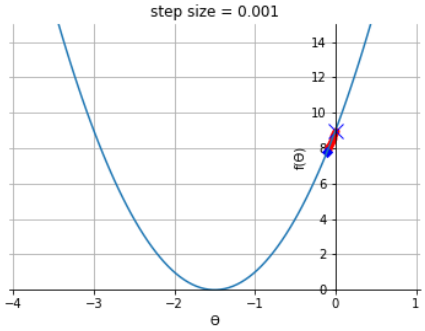
Remember, if it **diverges**, it never **approaches** a single value!




We start at the blue "x" mark. Notice that, even though we try to move toward the minimum, we go too far and accidentally get further and further!

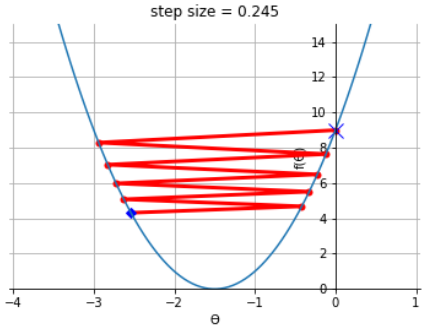
If they're too **small**, you might **converge** too slowly: it'll take way **too long** to make progress.

**Converging** means it successfully **approaches** an answer!



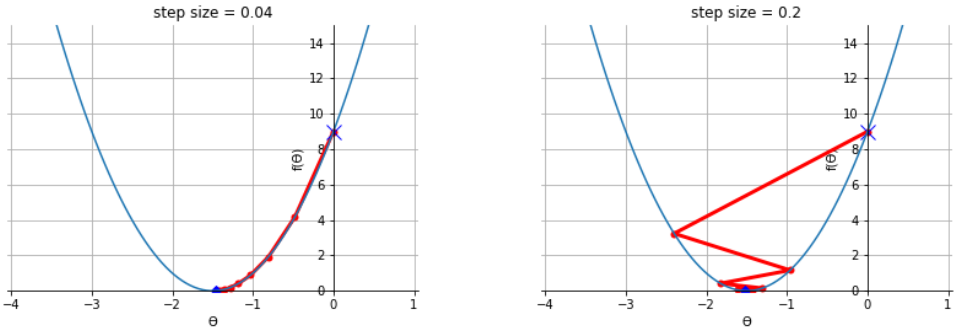

Our step size is too small: this is going to take too long!

In-between, it might converge, but **oscillate** a bunch: this can slow down getting an answer!




Most of our step is spent undoing the last step... we get better very slowly.

But, if we get the right step size, it'll converge nice and reasonably!




Both of these look pretty good! One of them oscillating a bit is fine.

One question you might ask is, "**how much** oscillation is too much? Am I converging **fast enough**?"

This is a good question, but the simple answer is that there is **no objective answer**: it depends on what you **need** and how much **time** you have. But you should strive to do **better** when you can!

**Concept 11**

Using the **wrong** step size can cause:

- Slow convergence
- Strong Oscillation
- Divergence

Which is why we **adjust** the step size using $\eta$.

### 3.1.6 Step size $\eta$

Right now, our step size is at the mercy of $J'(\theta)$. But, we don't have to be: we could **scale** our step size up or down.

We do this with our **scaling** factor (also called a **learning rate**), $\eta$.

So, we can rewrite our **change** in $\theta$ as:

$$\Delta\theta = -\eta \frac{dJ}{d\theta} = -\eta J'(\theta) \tag{3.9}$$

**Definition 12**

Our step size parameter $\eta$, or **eta**, **scales** how large each of our optimization steps are.

If $\eta$ is bigger, we might **learn** faster, but we also risk **diverging**.

Different values of $\eta$ are good for **different situations**.

### 3.1.7 Our procedure

So, we have our parameter **update**, $\Delta\theta$. We'll start at $t = 0$.

Before, we represented the $i^{th}$ **data point** with $x^{(i)}$. We'll reuse this **notation**.

**Notation 13**

Here, we're changing $\theta$ over **time**: each step happens at $t = \{1, 2, 3, \dots\}$ so we need **notation** for that.

We'll **reuse** the notation from $x^{(i)}$, for the $i^{th}$ data point.

In this case, we'll do $\theta^{(t)}$: the value of $\theta$ after $t$ **steps** are taken.

Earlier, we **introduced** $\theta_{old}$ and $\theta_{new}$: these are $\theta^{(t-1)}$ and $\theta^{(t)}$.

**Example:** After **10 steps** of 1-D gradient descent, we have gone from $\theta^{(0)}$ to $\theta^{(10)}$.

So, we move the **first** time using $J'(\theta^{(0)})$.

Once we've moved in parameter space **one** time, though, our **derivative** has changed: we're in a different part of the **surface**.

So, we'll take a **second** step with a **new** derivative, $J'(\theta^{(1)})$.

We want to do this **repeatedly**. We'll take our equation

$$\theta_{new} = \theta_{old} + \Delta\theta \tag{3.10}$$

And combine it with our **chosen** step size.

**Key Equation 14**

In **1-D**, **Gradient Descent** is implemented as follows:

At each time step t, we **improve** our hypothesis $\theta$ using the following rule:

$$\theta_{new} = \theta_{old} - \eta J'(\theta_{old})$$

Using $\theta^{(t)}$ notation:

$$\theta^{(t)} = \theta^{(t-1)} - \eta J'(\theta^{(t-1)})$$

We repeat until we reach whatever our chosen **termination condition** is.

We can also write it as:

$$\theta_{new} = \theta_{old} - \eta \cdot \left( \frac{dJ}{d\theta}\bigg|_{\theta=\theta_{old}} \right)$$

We've got our gradient descent **update** rule in 1-D!

### 3.1.8 Termination Conditions

When do we **stop**? We can't let it run forever.

We have some options:

- Stop after a **fixed** T steps.
  - This has the advantage of being **simple**, but how do you know what the **correct** number of steps is?
- Stop when $\theta$ **isn't changing** much: $|\Delta\theta| < \epsilon$, for example.
  - If our $\theta$ isn't changing much, our algorithm isn't **improving** our hypothesis much. So, it makes sense to stop: we've stabilized.
- Stop when the **derivative is small**: $|J'(\theta)| < \epsilon$.
  - Mathematically **equivalent** to our last choice. But a different **perspective**: if the slope is small, our surface is relatively **flat**, and we're near a **minimum** (probably).
  - "The derivative is **small**" is weaker, but in the same spirit as "the derivative is **zero**", $J'(\theta) = 0$, from last chapter.

### 3.1.9 Convergence Theorem

It turns out, if our function is **nice** enough, and we pick the **right** value of $\eta$, we can guarantee convergence!

**Theorem 15**
**Gradient Descent Convergence Theorem**

We want to optimize function J. If J is

- Smooth enough
- Convex

And

- $\eta$ is small enough

Then gradient descent **will** converge to the **global minimum**!

- "Small enough" seems vague, but it basically means, "if the first **two statements** are true, then there **exists** a choice of $\eta$ that converges."

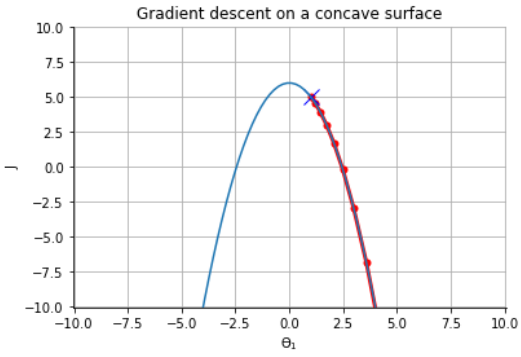
Or, if your $\eta$ is too **big**, you can keep trying **smaller** ones, until it works.

This is amazing! We can **guarantee** a best solution in some cases!

### 3.1.10 Concavity

One requirement we haven't focused on "J is **convex**". Why do we need J to be convex?

Well, if it's **concave**, there is no **global minimum**: it goes down forever!




Our gradient just leads us downhill forever.

**Concept 16**

If our function J is **concave**, then our result will not **converge**: it will continue to **decrease** more and more indefinitely.

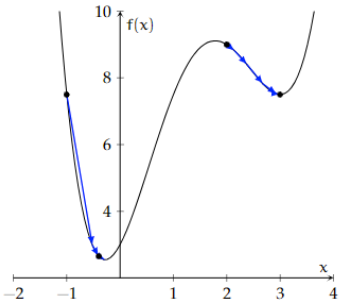
So, for future problems, let's assume it **doesn't** go down forever: if it was, then there is no best solution! We don't have a **valid** problem.

### 3.1.11 Local minima

Even if we don't have that problem, we have a **different** one:

Gradient descent **gradually** improves our solution until it reaches one it's **satisfied** with. But, what if there are **multiple** solutions we could reach?

Are they all equally good?




Depending on your starting position (**initialization**), you could find a different local minimum!

Maybe not! So, if our function isn't **always convex**, we can end up with **multiple** "valleys", or **local** minima.

**Definition 17**

A **global** minimum is the **lowest** point on our entire function: the one with the lowest **output**.

A **local** minimum is one that is the **lowest** point among those points that are **near** it.

- For **local minima**, if you add or subtract a **tiny** amount $\epsilon$ to the input, the output will **increase**.

So, we **won't** necessarily end up with the **global** minimum, even with a *small* $\eta$.

This shows that **initialization matters**!

**Definition 18**

**Initialization** is our "starting point": when we first **start** our algorithm, what are our **parameters** set to?

If we have a **different** starting position, we can find a **different** local minimum.

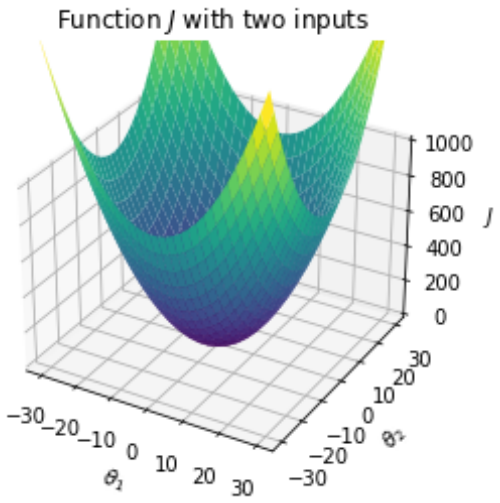
**Concept 19**

**Gradient descent** often finds **local** minima near the initialization, not necessarily **global** minima.

This means, if our function has **multiple local minima** (not fully convex), our **initialization** can affect our **solution**.

## 3.2 Multiple Dimensions

Now that we've handled the 1-D case, we'll move into 2-D: now, we have **two** parameters, $\theta_1$ and $\theta_2$, as the input to J.




The "height"of your plot in 3D, is, again, your output! You want to move **downhill**.

### 3.2.1 Multivariable Local Approximation (Review)

Again, we rely on **calculus**. We want to move up to having more parameters: more **dimensions**.

Before, in 1-D, we found that, if you **zoomed** in enough on a function (using a "**local** view"), we could **approximate** it as a **straight line**, and move up or down that slope.

There are **two** ways we can view our **approximation** in 2-D:

- First, we could turn it back into 1-D: we remove one variables.

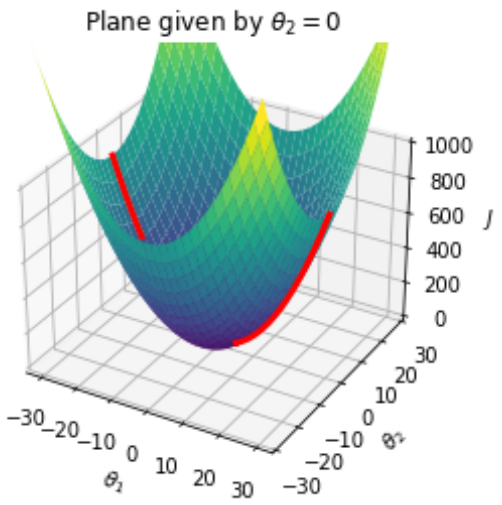
  We do this by turning one variable constant: take $\theta_2 = 0$. Now, we have one free variable $\theta_1$. Same as 1-D.

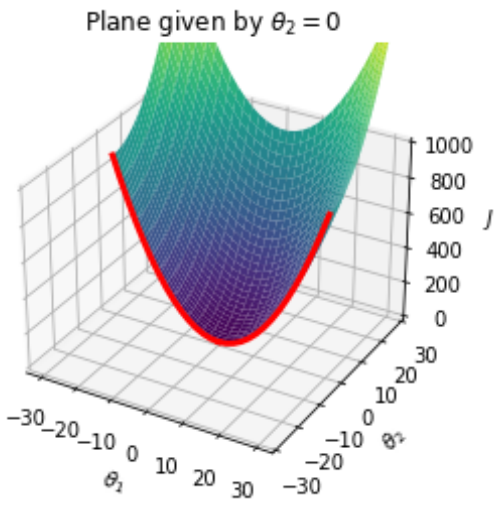
Remember that, by 2-D, we mean two **parameters**/inputs to J. If we add in the **height** of our function, that means our plot will **look** like 3-D!

**Concept 20**

We can **reduce** the number of **variables** we have to work with, by holding some of them **constant**. That way, we have a **simpler** problem to work with.

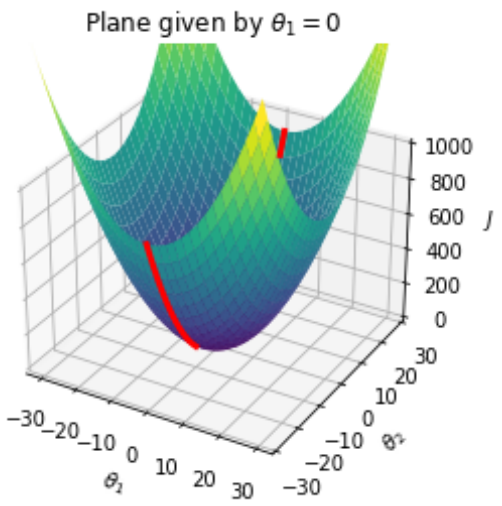
This is **the same** as taking a single 2-D plane in a 3-D plot.

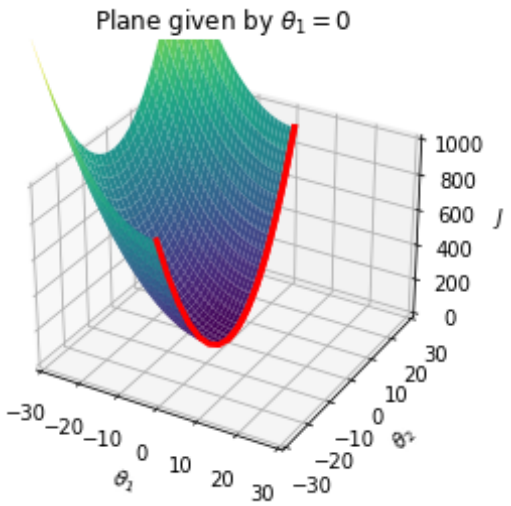



If we focus on a single plane of this surface, we end up with a **parabola**.

We can do the same the other way: we take $\theta_1 = 0$, and now we have a 1-D problem in $\theta_2$.




We can slice along the other axis as well!

Along each **axis**, $\theta_1$ and $\theta_2$, you can **approximate** our function as **two** different straight lines. Which leads into our next point...

- Second way: if we take the two perpendicular **lines** we got from each dimension, we can combine them into a **plane**.

**Concept 21**

If we have **two input variables** (a 2-D problem), we can **approximate** our surface as a **plane** if we **zoom** in enough.

If you look closely enough at any smooth surface in 3D space, it will look roughly "flat".

**Example:** The earth tends to look flat up close, even though it's a sphere.

These **approximations** will allow us to **optimize**.

### 3.2.2 2-D: One dimension at a time

How do we **improve** our function J? Now that we have **two** dimensions, we have to store our change $\Delta\theta$ in a **vector**:

$$\Delta\theta = \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix} \tag{3.11}$$

This **complicates** things: we have two different things to consider **at once**.

Well, the **simplest** way would be to treat it as a **1-D** problem, and do exactly what we did **before**.

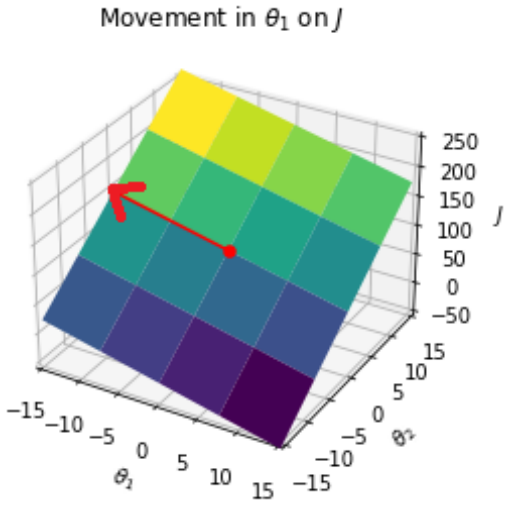
Note that we switched to **partial** derivatives, because we have **multiple** input variables $\theta_i$.

$$\Delta\theta_1 = \frac{\partial J}{\partial \theta_1} \tag{3.12}$$

Writing this in our **new** notation, we get:

$$\Delta\theta = -\eta \begin{bmatrix} \partial J/\partial\theta_1 \\ 0 \end{bmatrix} \tag{3.13}$$

And then we would take a **step**, moving along the $\theta_1$ **axis**.

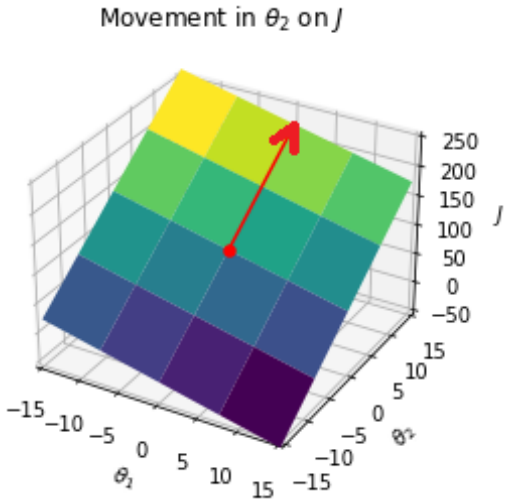



We can move along $\theta_1$ just like on a line.

What if we treated this as a 1-D problem for the **other** variable, $\theta_2$?

$$\Delta\theta = -\eta \begin{bmatrix} 0 \\ \partial J/\partial\theta_2 \end{bmatrix} \tag{3.14}$$

With this equation, we would be **moving** along the $\theta_2$ axis.




We can do the same with $\theta_2$.

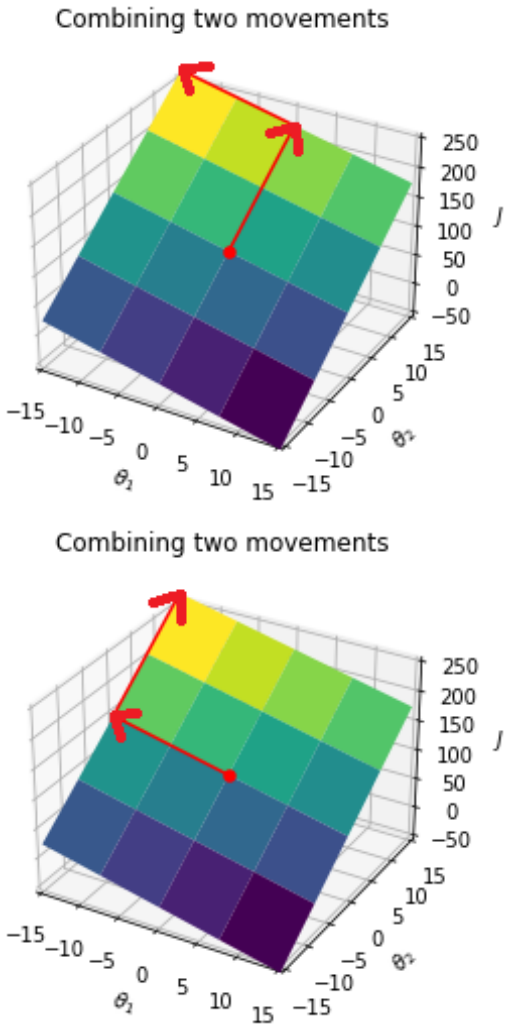
Why not move in **both** directions **at once**? We can **combine** our two derivatives: we'll add up our two steps.

**Linearity** means that I can **add** them up without anything **weird** happening.

$$\Delta\theta = -\eta \begin{bmatrix} \partial J/\partial\theta_1 \\ 0 \end{bmatrix} - \eta \begin{bmatrix} 0 \\ \partial J/\partial\theta_2 \end{bmatrix} \tag{3.15}$$

The relevant linearity rule: $L(x + y) = L(x) + L(y)$. In other words: taking two separate steps is the same as one big step.

These can be combined because we're treating our function as a **flat** plane: if I move in the $\theta_1$ direction first, it doesn't change the $\theta_2$ slope, and vice versa.




Our plane being flat means we can take both operations, back-to-back! Notice that the order doesn't matter.

$$\Delta\theta = -\eta \begin{bmatrix} \partial J/\partial \theta_1 \\ \partial J/\partial \theta_2 \end{bmatrix} \tag{3.16}$$

So, let's use that to optimize:

**Key Equation 22**

In **2-D**, you can optimize your function J using this rule:

$$\theta_{new} = \theta_{old} - \eta \underbrace{\begin{bmatrix} \partial J/\partial \theta_1 \\ \partial J/\partial \theta_2 \end{bmatrix}}_{\text{Using } \theta_{old}}$$

This is our **gradient descent** rule for 2-D.

This sort of approach makes some **sense**: if $\frac{\partial J}{\partial \theta_1}$ is **bigger** than $\frac{\partial J}{\partial \theta_2}$, that means that you can get **more benefit** from moving in the $\theta_1$ direction than $\theta_2$.

So, in that case, your step will move more in the $\theta_1$ direction: it's a more **efficient** way to get a **better** hypothesis!

But for now, we **don't know** that this is necessarily the **optimal** way to change $\theta$ - we'll explore that later.

### 3.2.3 Gradient Descent in n-D

This idea can be built up in **any number** of dimensions: each variable $\theta_k$ creates a **different** line we can use to **approximate**.

And, we can combine them into a **flat** **hyperplane** : so, we can **add up** all of the different **derivatives**.

A hyperplane is just the equivalent of a plane in a higher dimensional space.

In 3-D space, a plane fills one "slice" of 2-D space. In 4-D space, the hyperplane fills up one "slice" of 3-D space.

**Key Equation 23**

In **n-D**, you can optimize your function J using this rule:

$$\theta_{new} = \theta_{old} - \eta \underbrace{\begin{bmatrix} \partial J/\partial \theta_1 \\ \partial J/\partial \theta_2 \\ \vdots \\ \partial J/\partial \theta_d \end{bmatrix}}_{\text{Using } \theta_{old}}$$

This is our **generalized gradient descent** rule.

### 3.2.4 The Gradient

We call this **gradient** descent because that right term we just invented **is** the gradient!

**Definition 24**

The gradient can be written as

$$\nabla_\theta J = \begin{bmatrix} \partial J/\partial \theta_1 \\ \partial J/\partial \theta_2 \\ \vdots \\ \partial J/\partial \theta_d \end{bmatrix} = \frac{dJ}{d\theta}$$

So, our rule can be rewritten (for the last time) as:

**Key Equation 25**

The **gradient descent** rule can be generally written as:

$$\theta_{new} = \theta_{old} - \eta \nabla_\theta J(\theta_{old})$$

$\theta_{old}$ is the input to $\nabla_\theta J$, not multiplication!

$$\theta_{new} = \theta_{old} - \eta J'(\theta_{old})$$

Now, is $\nabla_\theta J$ the **optimal** way to improve(optimize) $\theta$? Let's find out.

### 3.2.5 The Plane Approximation

So, what **is** the best direction? Which way will increase/decrease J **fastest?**

*Is* it the gradient? Let's explore a bit to figure that out. Let's look at our plane, and see what hints it might provide:

For explanation purposes, we'll assume 2-D, but the explanation extends to n-D.

**Concept 26**

Assume your function is, at least locally, a **flat plane**.

- A **flat plane** has only **one** direction of **maximum increase**: this is the direction you might call, "directly **uphill**" if you think of elevation.
- The **opposite** direction is the direction of **maximum decrease**, or "**downhill**".
- If you move at a **right angle** to the "best" direction (maximum increase/decrease), the function **will not change**. In elevation, you stay at the **same height**!

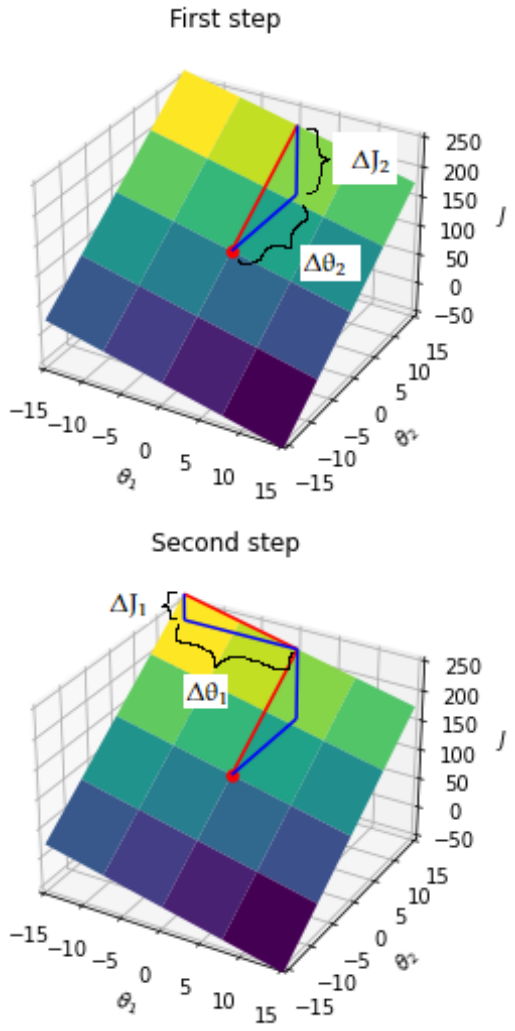
This is useful! We can **break down** any direction into the part that **affects** our function J, and the part that **doesn't**.

In the n-D case, we have **more** perpendicular directions. But, all of them have **no effect**!

### 3.2.6 The Optimal Direction: The Gradient

How do we get the optimal direction?

The **total** change in J is gotten by just **adding** the change in each direction (planes are simple, which makes this possible!):




You can add up the results of our two steps: $\Delta J_2$ and $\Delta J_1$.

$$\Delta J \approx \Delta J_1 + \Delta J_2 \tag{3.17}$$

Let's convert that using derivatives:

$$\Delta J \approx \Delta\theta_1 \frac{\partial J}{\partial \theta_1} + \Delta\theta_2 \frac{\partial J}{\partial \theta_2} \tag{3.18}$$

Now we've got a useful equation: the total change. As a bonus we can see a clear **pattern**

($\Delta\theta_i$ matches $i^{th}$ derivative).

So, **condense** this pattern, like we did for our linear model: using a **dot product**.

$$\Delta J \approx \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix} \cdot \begin{bmatrix} \partial J/\partial\theta_1 \\ \partial J/\partial\theta_2 \end{bmatrix} = \Delta\theta \cdot \nabla_\theta J \tag{3.19}$$

The **gradient** shows up! Interesting. But what does that **mean**?

Well, we want to **maximize** (or minimize!) our $\Delta J$. How do we maximize a **dot product**?

- A dot product $a \cdot b$ is maximized when the directions of $a$ and $b$ are **the same**!
- The direction of the gradient was given to us by our calculations.
- So, we want $\Delta\theta$ to match the **gradient**! That way, they're in the same direction.

Maximizing this dot product means maximizing $\Delta J$, which is our goal.

So, we just demonstrated that the **gradient** gives us the **best** direction for $\Delta\theta$.

So, all we have to do is to **flip** the sign to **minimize** $\Delta J$.

And so, gradient descent is complete!

**Concept 27**

The **gradient** $\nabla J$ is the **direction of greatest increase** for $J$.

That means means the opposite direction $-\nabla J$ is the **direction of greatest decrease** in $J$.

This is the single **most important concept** in this entire chapter!

### 3.2.7 Termination Condition

We can still use our termination conditions from before, but we need to be careful to make sure they extrapolate to n-D.

- Stop after a fixed T steps.
  - Nothing to change here.

We don't use this one often, though!

- Stop when $\|\theta\|$ isn't changing much: $\|\Delta\theta\| < \epsilon$, for example.
  - We just had to replace **absolute** value with **magnitude**.
- Stop when the derivative is small: $|J'(\theta)| < \epsilon$
  - Nothing to change here.

### 3.2.8 Another explanation of gradient (OPTIONAL)

Some students may not like the first explanation given for why gradient is the **direction of greatest increase**. So here, we use a slightly **different** approach, one that's more **geometric**.

Feel free to skip this section if you are not interested.

We look at a random 2-D vector, $\Delta\theta$ - no assurances about how good or bad it is.

Currently, our vector **components** are based on $\theta_1$ and $\theta_2$. But, it can be useful to **switch** perspectives.

Our vector can **also** be broken up into **parts** based on whether it **affects** J. This will let us take a **look** at the "best direction" we're trying to **find**.

- Uphill: the "best" direction $\hat{u}_{best}$ (magnitude $\Delta B$)
- Same height: the direction with no effect, $\hat{u}_{none}$ (magnitude $\Delta N$)

As we established before, these two directions are perpendicular on the plane.

What about higher dimensions?

If we have k more dimensions, we just add k more unit vectors which add nothing to $\Delta J$.

$$\Delta\theta = \underbrace{u_{best}}_{\text{Full effect on J}} + \underbrace{u_{none}}_{\text{No effect on J}} = \Delta B * \hat{u}_{best} + \Delta N * \hat{u}_{none} \tag{3.20}$$

So, all of the change in J just comes from $u_{best}$. We **don't care** about the other direction!

**Concept 28**

In a local planar approximation, the **only** component of $\Delta\theta$ that **affects** J is the **direction of greatest increase**, $u_{best}$.

So, we can determine $\Delta J$ using **only that component**.

Thus, $u_{best}$ gives us the "direction of greatest increase": if we rotate our vector, we replace some of $u_{best}$ with $u_{none}$.

- In which case, we're losing some $\Delta J$. So, we don't want to change direction like that.

However, this is the same kind of behavior as the dot product:

$$\Delta J \approx \Delta\theta \cdot \nabla_\theta J \tag{3.21}$$

- When we do the dot product $a \cdot b$, we take the **projection** of a onto b: we only take the component of a in the same direction as b.
- In this case, we ony include the **component** of $\Delta\theta$ which matches $\nabla_\theta J$.

So, let's compare the two.

- The $u_{best}$ component of $\Delta\theta$ is the only part that matters for $\Delta J$.
- The $\nabla_\theta J$ component of $\Delta\theta$ is the only part that matters for the **dot product**, which equals $\Delta J$.

They're the same!

## 3.3 Application to Regression

One nice thing about **gradient descent** is that it is **easy** to switch the kind of problem you're applying it to: all you need is your **parameters**(s) $\theta$, and a function to optimize, J.

From there, you can just **compute** the gradient.

### 3.3.1 Ordinary Least Squares

Our **loss** function is

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left( (\theta^T x^{(i)} + \theta_0) - y^{(i)} \right)^2 \tag{3.22}$$

Or, in **matrix** terms, Including the appended row of 1's from before.

$$J = \frac{1}{n} \left( \tilde{X}\theta - \tilde{Y} \right)^T \left( \tilde{X}\theta - \tilde{Y} \right)$$

Our gradient, according to **matrix derivative** rules, is

$$\nabla_\theta J(\theta) = \frac{2}{n} \tilde{X}^T \left( \tilde{X}\theta - \tilde{Y} \right) \tag{3.23}$$

Before, we set it equal to **zero**. But here, we can instead take **steps** towards the solution, using **gradient descent**.

We could use the **matrix** form, but sometimes it's easier to use a **sum**. Fortunately, derivatives are easy with a sum. If so, here's **another** way to write it:

$$\nabla_\theta J(\theta) = \frac{2}{n} \sum_{i=1}^{n} \left( \theta^T x^{(i)} - y^{(i)} \right) x^{(i)} \tag{3.24}$$

Either way, we use gradient descent **normally**: Remember that $\theta_{old}$ is an **input** to the gradient, not multiplied by it!

$$\theta_{new} = \theta_{old} - \eta \nabla_\theta J(\theta_{old})$$

Using $\theta^{(t)}$ notation:

$$\theta^{(t)} = \theta^{(t-1)} - \eta \nabla_\theta J(\theta^{(t-1)})$$

### 3.3.2 Ridge Regression

Ridge regression is similar.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}}\right)^2 + \underbrace{\lambda\|\theta\|^2}_{\text{Regularizer}}$$

However, we have to treat $\theta_0$ as **separate** from our other data points, because of **regularization**: remember that it **doesn't** apply to $\theta_0$.

For $\theta$:

$$\nabla_\theta J_{\text{ridge}}(\theta, \theta_0) = \frac{2}{n}\sum_{i=1}^{n}\left((\theta^T x^{(i)} + \theta_0) - y^{(i)}\right)x^{(i)} + 2\lambda\theta \tag{3.25}$$

For $\theta_0$:

$$\frac{\partial J_{\text{ridge}}(\theta, \theta_0)}{\partial \theta_0} = \frac{2}{n}\sum_{i=1}^{n}\left((\theta^T x^{(i)} + \theta_0) - y^{(i)}\right) \tag{3.26}$$

Notice that we used a **gradient** for our vector $\theta$, but since $\theta_0$ is a single variable, we just used a **simple derivative**!

**Concept 29**

The **gradient** $\frac{dJ}{d\theta}$ must have the **same shape as** $\theta$: this shape-matching is why we can easily **subtract** it during gradient descent.

$$\underbrace{\theta_{new}}_{(d\times 1)} = \underbrace{\theta_{old}}_{(d\times 1)} - \eta \underbrace{\nabla_\theta J(\theta_{old})}_{(d\times 1)}$$

The derivative $\frac{dJ}{d\theta_0}$ is based on $\theta_0$, a constant. So, the shape must be $(1 \times 1)$.

$$\theta_0^{(t)} = \theta_0^{(t-1)} - \eta \left.\frac{dJ}{d\theta_0}\right|_{\theta_0 = \theta_0^{(t-1)}}$$

### 3.3.3 Computational Gradient

Sometimes, we **can't** easily find the **equation** for our gradient: maybe our loss isn't a simple **equation**, or we have some **other** kind of problem. So, rather than getting the **exact** gradient, we **approximate** it.

But how do we **approximate** the gradient? Well, first, we could **reference** how we approximate a **simple derivative**.

A derivative is just a 1-D gradient, after all!

The definition of the **derivative** can be gotten as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3.27}$$

But, what if we can't take the **limit**? Or, we just don't **want** to?

We can **approximate** by taking $h$ to be a small, **finite** number.

Instead of $h$, we'll call this $\delta$.

**Concept 30**

When **approximating** the derivative, we can choose a **small** finite width to measure, called $\delta$, so that

$$\frac{df}{dx} \approx \frac{f(x+\delta) - f(x)}{\delta}, \qquad \delta << 1$$

So, let's **extend** that to the **gradient**:

$$\nabla_\theta J = \begin{bmatrix} \partial J/\partial \theta_1 \\ \partial J/\partial \theta_2 \\ \vdots \\ \partial J/\partial \theta_d \end{bmatrix} \tag{3.28}$$

Luckily, the **gradient** is just a bunch of derivatives **stacked** in a **vector**!

So, we can just **compute** each of them **separately**, and then put them together.

Let's show how we'd **write** that in **vector** form, for just one of them. We want something like

$$\underbrace{J'(\theta) \approx \frac{J(\theta+\delta) - f(\theta)}{\delta}}_{\text{Not correct, but closer}} \tag{3.29}$$

This isn't quite right, because a **scalar** $\delta$ would **add** to **every term**.

We **only** want to shift **one** variable at a time, so we can do a **simple** derivative.

Let's say we want $dJ/d\theta_1$. We would **only** want to add $\delta$ to $\theta_1$: the other parameters are **unchanged**.

So, we **can't** add a **scalar**. Instead, we need a $(d \times 1)$ vector: one term to **separately** add to each $\theta_k$ term.

$$\Delta\theta = \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \\ \vdots \\ \Delta\theta_d \end{bmatrix} \tag{3.30}$$

We want most terms **unchanged**, so we'll **add 0** to each of them, and we'll add $\delta$ to the one term we want to **edit**.

$$\Delta\theta = \begin{bmatrix} \delta \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{3.31}$$

We'll **create** one of these vectors for each **dimension**. We'll give them a special **name**: $\delta_k$, for the $k^{th}$ dimension.

$$\delta_1 = \begin{bmatrix} \delta \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad \delta_2 = \begin{bmatrix} 0 \\ \delta \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad \delta_{d-1} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \delta \\ 0 \end{bmatrix} \quad \delta_d = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \delta \end{bmatrix} \tag{3.32}$$

Finally, we'll **divide** by $\delta$. We have what we need for our full equation:

**Key Equation 31**

In order to **computationally find the gradient**, you need to find the **partial derivative** for each term $\theta_k$.

$$\frac{dJ}{d\theta_k} \approx \frac{J(\theta + \delta_k) - J(\theta)}{\delta}$$

Where

- $\delta$ is a small positive number
- $\delta_k$ is the $(d \times 1)$ **column vector** with a $\delta$ in the $k^{th}$ row, and a 0 in every other row.

### 3.3.4 Problems with Gradient Descent

Gradient descent is very handy, but it's important to be aware of some of its **problems**.

We've discussed a couple: diverging, oscillating, and converging slowly. We also have to

worry about **local minima** that aren't as good as other answers.

But there's also a **requirement**: our loss function has to be **smooth** and **differentiable**. If it isn't, we can't take the **gradient** of it.

**Concept 32**

**Gradient descent** requires for your **functions** to be (at least mostly) **smooth and differentiable.**

Our **answer** is also only as good as our **loss function**: if our loss function is not good for what we actually want to **accomplish**, then we can easily create a **bad** model.

## 3.4 Stochastic Gradient Descent

### 3.4.1 Another problem with gradient descent

Some **benefits** of gradient descent come from the fact that GD **gradually** improves:

- You can pause early to check progress, or quit early if you model is good enough. We save on computation time.

But we can improve this feature: currently, we use a **sum** of all data points to get our gradient. Meaning, we have to compute all of our data before we take a single step.

### 3.4.2 A better way: stochastic GD

Instead, why wait until we have **added** up over all the data? We could just **compute** the gradient over **one** data point **at a time**. In fact, to be fair, we'll do it **randomly**.

To compensate for using less data, our steps will have to be **smaller**!

But wait, this **seems** like it would be **less** effective - after all, how much does **one** data point tell you?

Well, even if it isn't much, this isn't very **different** from adding them up all at **once**: in **theory**, taking lots of **little** steps should average out to the **same** information as if we do it all at once.

**Definition 33**

**Stochastic Gradient Descent (SGD)** is the process of applying **gradient descent** on **randomly** selected data points.

This should **average** out to being **similar** to regular (batch) gradient descent, but the **randomness** often lets it improve **faster** and **avoid** some common problems.

Stochastic is just a very mathematically precise word for "random".

There are more possible benefits, too: **randomly** choosing data points adds some **noise**, and random movement might be able to pull us out of local minima we don't want.

We mean "noise" in the signals sense: random **variation** in our data. Randomly choosing data points is more unpredictable than using all of our data.

This sort of **noise** and **randomness** can make it hard for our model to **perfectly** fit the training data: this can reduce **overfitting**, too!

The random selection makes the data "look different" each time, so it's hard to perfectly match it.

**Concept 34**

There are many **benefits** to **SGD** (Stochastic Gradient Descent) over regular BGD (Batch Gradient Descent).

- SGD can sometimes **learn** a good model **without** using all of our **data**, which can **save us time** when data sets are **too large**.
  - It can also let us address problems **early** if the model **isn't** improving.
- The noise produced by the random sampling in SGD can sometimes help it **avoid local minima**.
  - This is because the model might be moved in a **random direction** in **parameter space**, and randomly **pulled out** of that minimum, even if BGD would have gotten **stuck**.
- The noise also **reduces overfitting**, because it's **harder** for the model to **memorize** the exact details of the **distribution**.

### 3.4.3 Ensuring Convergence

How do we make sure that our SGD method converges? We need some kind of termination criteria. Thankfully, there's a useful theorem on the matter:

**Theorem 35**

SGD **converges** with *probability one* to the **optimal** $\Theta$ if

- $f$ is convex

And our step size(learning rule) $\eta(t)$ follows these rules:

$$\sum_{t=1}^{\infty} \eta(t) = \infty \qquad \text{and} \qquad \sum_{t=1}^{\infty} \eta(t)^2 < \infty$$

Why these rules? Let's see:

- The **first** rule is for the **same** reason as for regular BGD: if it isn't **convex**, we can get stuck in **local minima**, or if it's **concave**, decrease **forever**.
- The **second** rule means that your steps need to add up to an **infinite distance**: this allows you to reach **any** possible point in your **parameter space**.
- The **third** one is a bit **trickier**, but basically means the steps need to get **smaller**, so we can approach the **minimum** (otherwise we might **diverge**!)

One option is $\eta(t) = 1/t$. But often, we use rules that **decrease** more **slowly**, so that it doesn't take as **long**.

In this case, though, we're often **no longer** guaranteed convergence.

Why? Because of our third condition, $\sum_t \eta(t) < \infty$. If $\eta(t)$ decreases too slowly, this sum goes to infinity, and our GD algorithm might **diverge**.

## 3.5 Terms

- Gradient Descent
- Parameter Space
- Local view (Calculus)
- Linear Function Approximation
- Planar Function Approximations
- Convergence
- Divergence
- Oscillation
- Step size
- Termination Condition
- Concavity/Convexity
- Global Minimum
- Local Minimum
- Initialization
- Gradient (Direction of Maximum Increase)
- Gradient Descent Rule
- Gradient Shape
- Gradient Approximation
- Stochastic Gradient Descent
- Batch Gradient Descent
- BGD Convergence Theorem
- SGD Convergence Theorem