

# Gradient Descent

Many problems in Machine Learning are framed as *optimization* problems

- Find the choice of parameters  $\Theta$
- That minimizes a Loss function

The best (optimal)  $\Theta$  is the one that minimizes the Average (across training examples) Loss

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}} \mathcal{L}_{\Theta}$$

Many Classical ML problems are designed such that  $\Theta^*$  has a closed-form solution

- Maximum likelihood estimates for Linear Regression

Closed form solutions, however, may only be feasible for Loss function of restricted form.

When a closed form solution is not possible, we may find  $\Theta^*$  via a search process known as Gradient Descent.

In the Deep Learning part of the course, virtually all Loss functions will require this form of solution.

## Loss functions, review

- $\hat{\mathbf{y}}^{(i)} = h(\mathbf{x}^{(i)}; \Theta)$ , the prediction for example  $\mathbf{x}^{(i)}$  with target label  $\mathbf{y}^{(i)}$
- Per-example loss

$$\mathcal{L}_{\Theta}^{(i)} = L( h(\mathbf{x}^{(i)}; \Theta), \mathbf{y}^{(i)} ) = L(\hat{\mathbf{y}}^{(i)}, \mathbf{y})$$

- The Loss for the entire training set is simply the average (across examples) of the Loss for the example

$$\mathcal{L}_{\Theta} = \frac{1}{m} \sum_{i=1}^m \mathcal{L}_{\Theta}^{(i)}$$

Two common forms of  $L$  are Mean Squared Error (for Regression) and Cross Entropy Loss (for classification).

# Optimization

How do we find the  $\Theta^*$  that minimizes  $\mathcal{L}$ ?

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}} \mathcal{L}_{\Theta}$$

One way is via a search-like procedure known as Gradient Descent:

We start with an initial guess for  $\Theta$  and then:

- Evaluate  $\mathcal{L}_\Theta$  across training examples
$$\langle \mathbf{X}, \mathbf{y} \rangle = [\mathbf{x}^{(i)}, \mathbf{y}^{(i)} | 1 \leq i \leq m]$$
- Make a *small change* to  $\Theta$  that results in a reduced  $\mathcal{L}_\Theta$
- Repeat until  $\mathcal{L}_\Theta$  stops decreasing

Fortunately, for many functions  $\mathcal{L}_\Theta$  we can use calculus to guide the small change in  $\Theta$  in the direction of reduced  $\mathcal{L}_\Theta$

$$\frac{\partial}{\partial \Theta} \mathcal{L}_\Theta$$

is the partial derivative of  $\mathcal{L}_\Theta$  with respect to  $\Theta$ .

- For a unit increase in  $\Theta$ :  $\mathcal{L}_\Theta$  *increases* by  $\frac{\partial}{\partial \Theta} \mathcal{L}_\Theta$

Thus, to *decrease*  $\mathcal{L}_\Theta$  we only need to add an increment in  $\Theta$  proportional to the *negative* of the partial derivative.

Since  $\Theta$  is a vector, the partial derivative is *also* a vector and is called the *gradient*.

The iterative process we described is called *gradient descent* as it follows the negative of the gradient towards a minimum for  $\Theta$ .



Here is some illustrative code we will use to demonstrate the process.

```
In [5]: def f(x):
        return x**2

def deriv(f, x_0):
    h = 0.000000001 #step-size
    return (f(x_0 + h) - f(x_0))/h

def tangent(f, x_0, x=None):
    y_0 = f(x_0)
    slope = deriv(f, x_0)

    if x is not None:
        r = 2
        xmin, xmax = np.min(x), np.max(x)
        xlo, xhi = max(x_0 - r, xmin), min(x_0 + r, xmax)
    else:
        r = 2
        xlo, xhi = x_0 - r, x_0 + r

    xline = np.linspace(xlo, xhi, 10)
    yline = y_0 + slope*( xline - x_0)

    return xline, yline

def plot_tangent(f, x_s, x, ax, show_tangent=True):
    # Plot function
    _ = ax.plot(x, f(x))

    # Plot tangent point x_s
    y_s = f(x_s)
    ax.scatter(x_s, y_s, color='r', s=90)

    # Plot tangent line
    if show_tangent:
        xtang, ytang = tangent(f, x_s, x)
        ax.plot(xtang, ytang, 'g--')
```

```
return ax
```

```
In [6]: def plot_step(f, x_s, x, show_tangent=True, visible=True):
        fig, ax = plt.subplots(1, 1, figsize=(12,6))

        y_s = f(x_s)

        # Plot the function, the point, and optionally: the tangent line
        _ = plot_tangent(f, x_s, x, ax, show_tangent=show_tangent)

        _ = ax.set_xlabel("$\Theta$", fontsize=16)
        _ = ax.set_ylabel("$L$", fontsize=16, rotation=0)

        if not visible:
            plt.close(fig)

        return fig, ax
```

```

In [7]: def plot_gradient_descent(max_steps=4, alphas=[ 0.1, 0.4, 0.7, 1.0 ]):

    fig, axs = plt.subplots(len(alphas), max_steps, figsize=(20,min(12, 6 * len
(alphas))))
    axs = axs.reshape( (len(alphas), max_steps) ) # Take care of special case w
here len(alpha) == 1

    for a_idx, alpha in enumerate(alphas):
        x_s = x_0

        for step in range(0,max_steps):
            ax = axs[a_idx, step]
            _ = ax.set_xlabel(" $\theta$ ", fontsize=16)
            _ = ax.set_ylabel(" $L$ ", fontsize=16, rotation=0)
            _ = ax.set_title('$\\alpha$={a:3.2f}'.format(a=alpha))
            y_s = f(x_s)

            # Obtain tangent line at x0
            _ = plot_tangent(f, x_s, x, ax)

            # Update x_s
            slope = deriv(f, x_s)
            x_s = x_s + alpha * (- slope)

        _ = fig.tight_layout()

    plt.close(fig)
    return fig, axs

```

```
In [8]: alpha = 0.4  
x = np.linspace(-5, +5, 30)
```

# Gradient Descent: Overview

Let's illustrate the process with an example

Let's plot a simple loss function as an illustration.

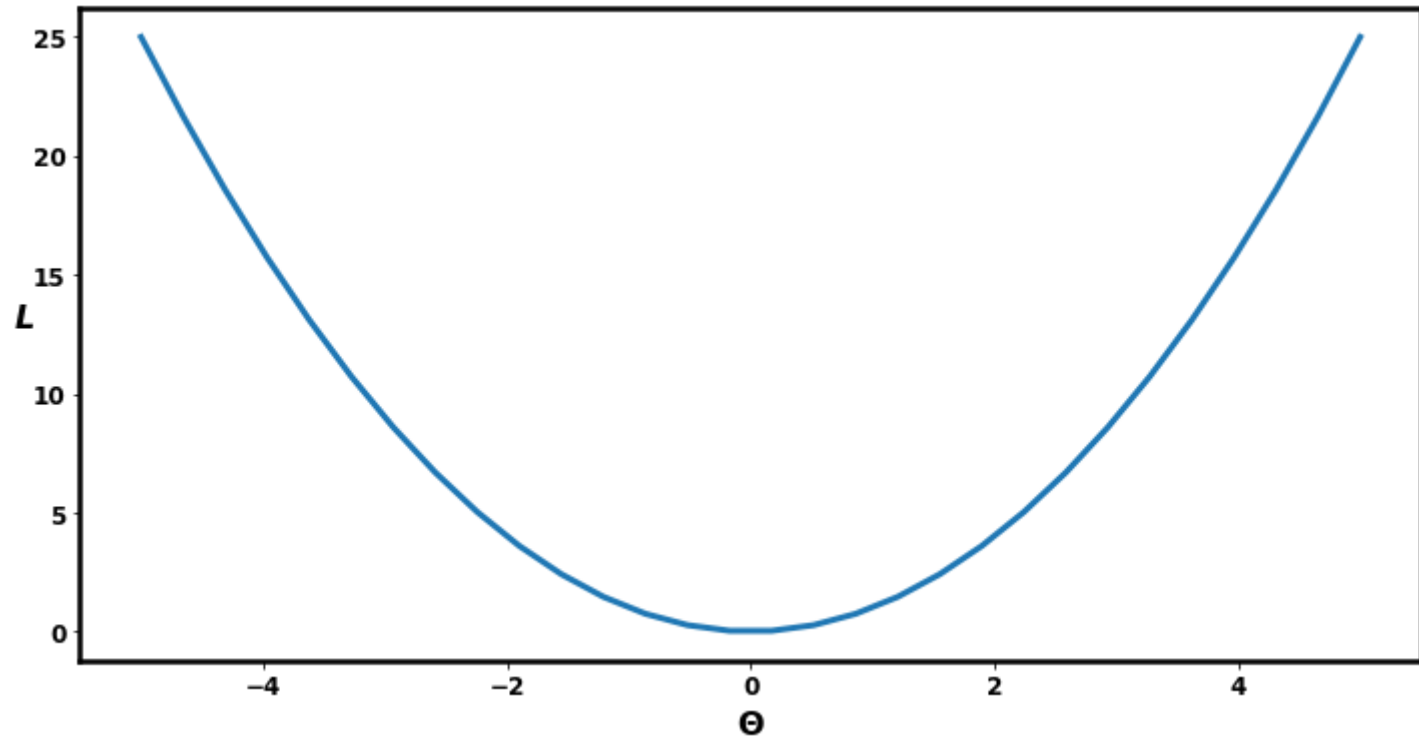
In this simple example:  $\Theta$  is a vector of length 1.



```
In [9]: fig, ax = plt.subplots(1,1, figsize=(12,6))
        _ = ax.plot(x, f(x), linewidth=3)
        _ = ax.set_xlabel("$\Theta$", fontsize=16)
        _ = ax.set_ylabel("$L$", fontsize=16, rotation=0)
        plt.close(fig)
```

In [10]: fig

Out[10]:



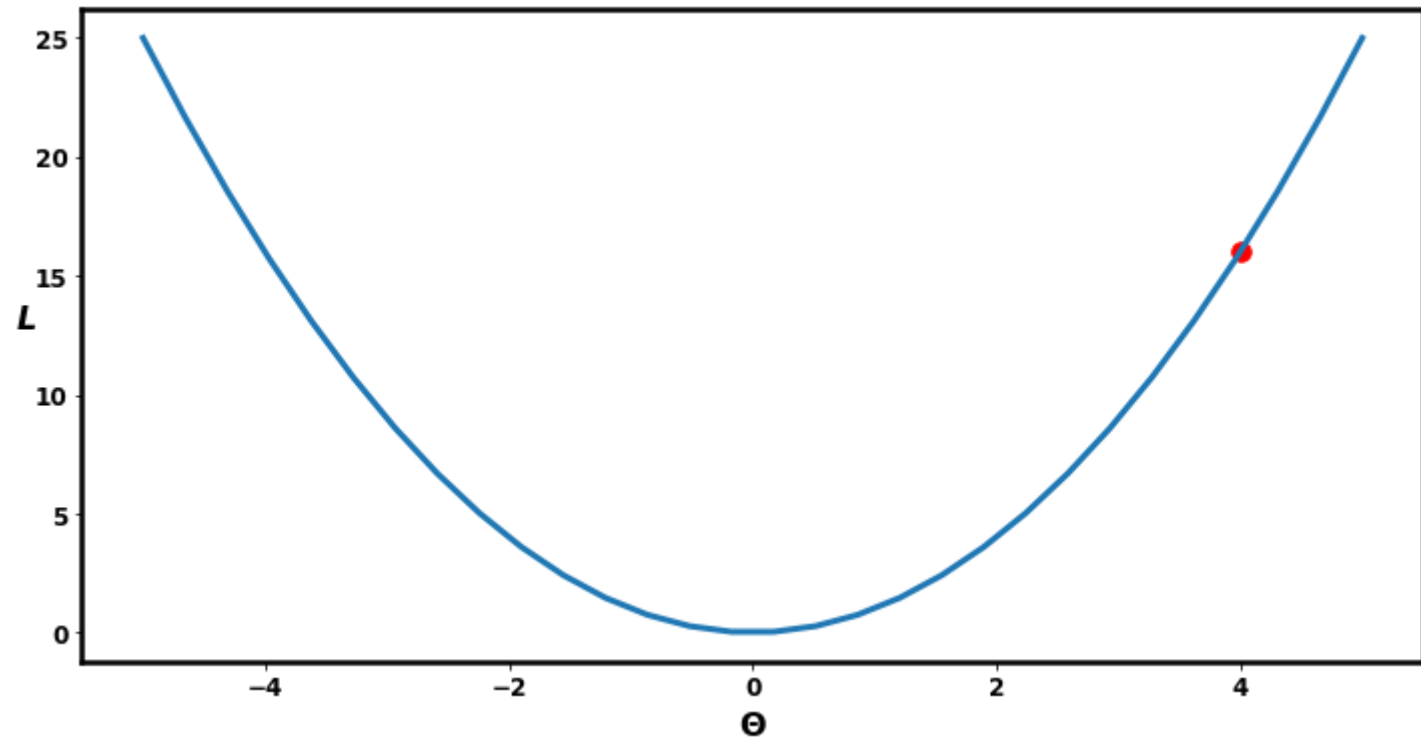
Let's start off with a guess for  $\Theta$

```
In [11]: x_0 = 4  
         x_s = x_0
```

```
In [12]: fig, ax = plot_step(f, x_s, x, show_tangent=False, visible=False)
```

In [13]: fig

Out[13]:



Clearly not at a minimum.

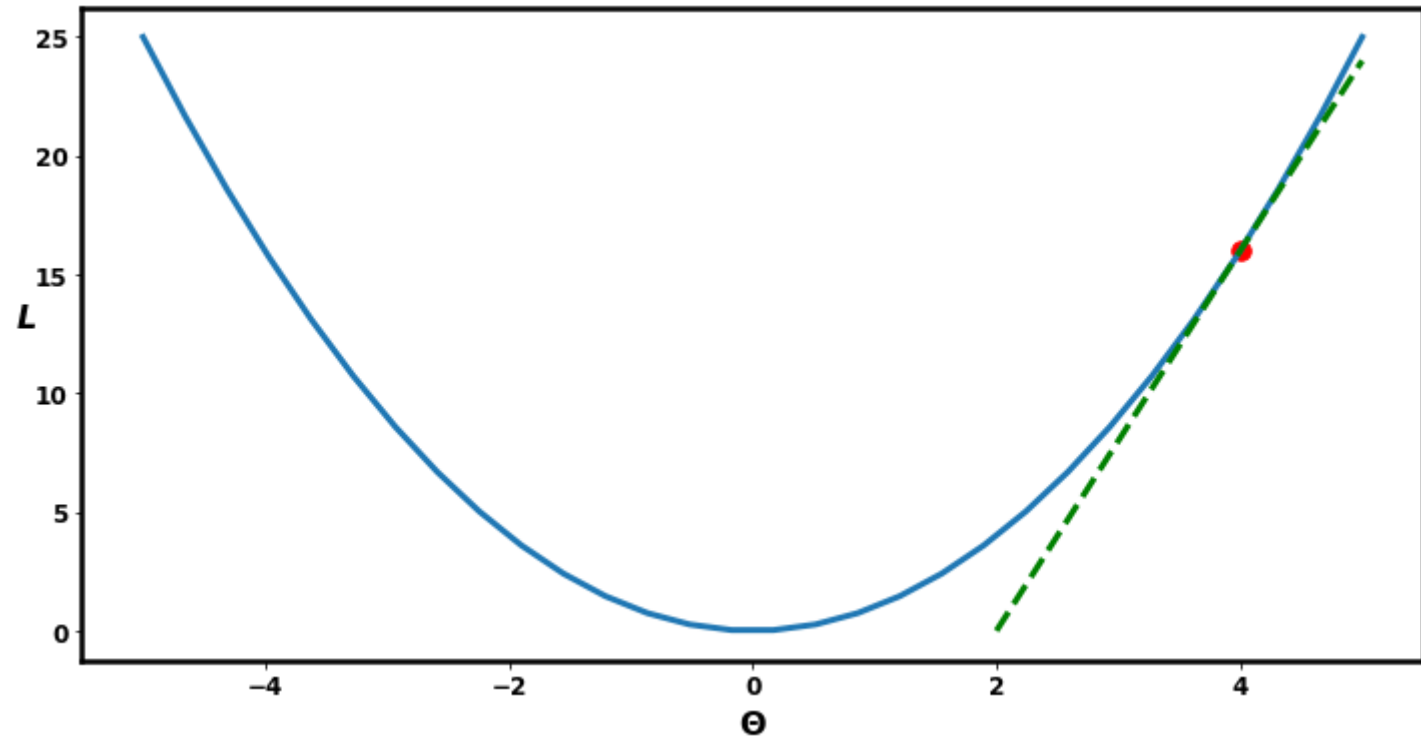
Compute the gradient of  $\mathcal{L}_\Theta$  at initial guess  $x_s$

```
In [14]: x_s = x_0  
fig, ax = plot_step(f, x_s, x, visible=False)
```



In [15]: fig

Out[15]:



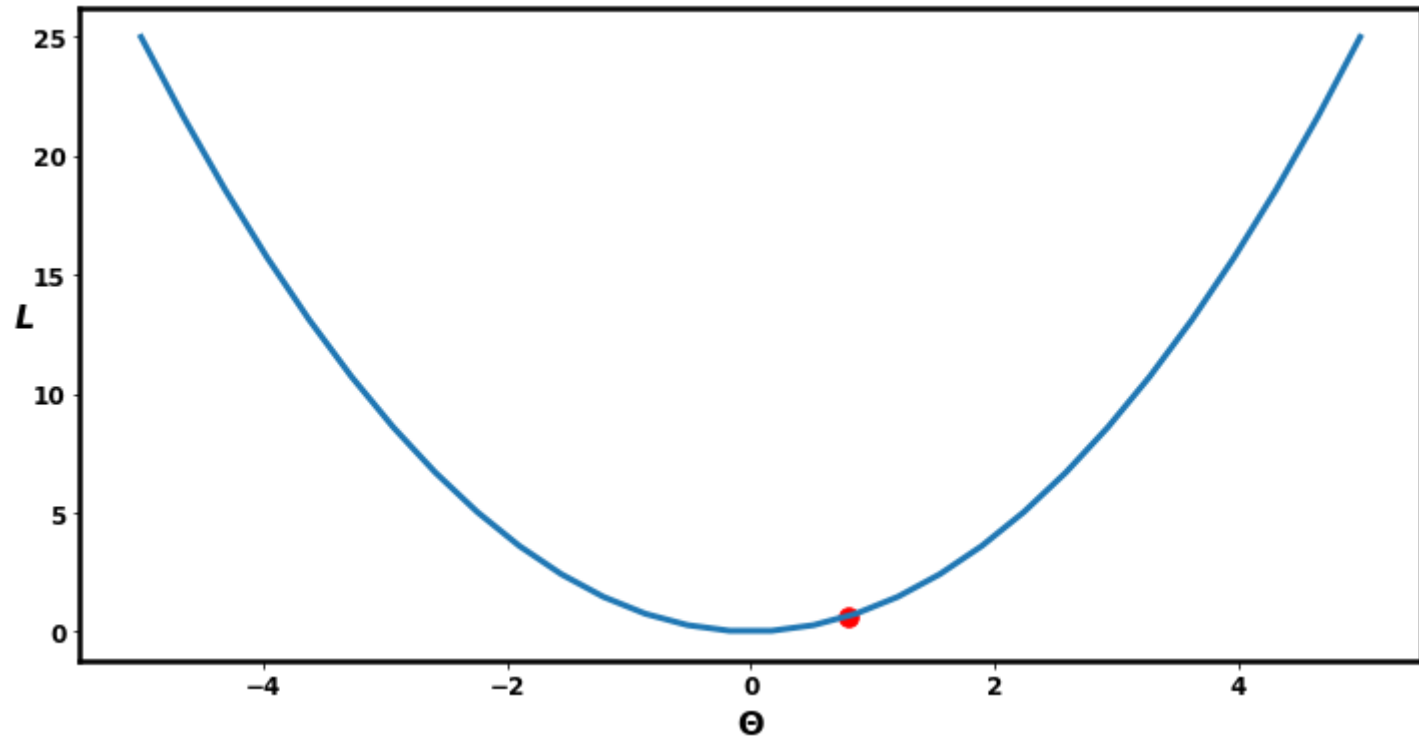
Let's modify our guess by moving in proportion ( $\alpha$ ) to the negative of the gradient:

```
In [16]: # Update x_s  
slope = deriv(f, x_s)  
x_s = x_s + alpha * (- slope)
```

```
In [17]: fig, ax = plot_step(f, x_s, x, show_tangent=False, visible=False)
```

In [18]: fig

Out[18]:



By following the gradient as we did: we wind up at a new  $\Theta$  where  $\mathcal{L}_{\Theta}$  is reduced compared to that at the original guess.

Taking the gradient of the  $\mathcal{L}$  at the new point, we continue the iterative process.

```
In [19]: if CREATE_MOVIE:  
         _= gdh.create_gif2(x, f, x_0, out="images/gd.gif", alpha=alpha)
```

```
In [32]: _ = gdh.display_gif("images/gd.gif")
```

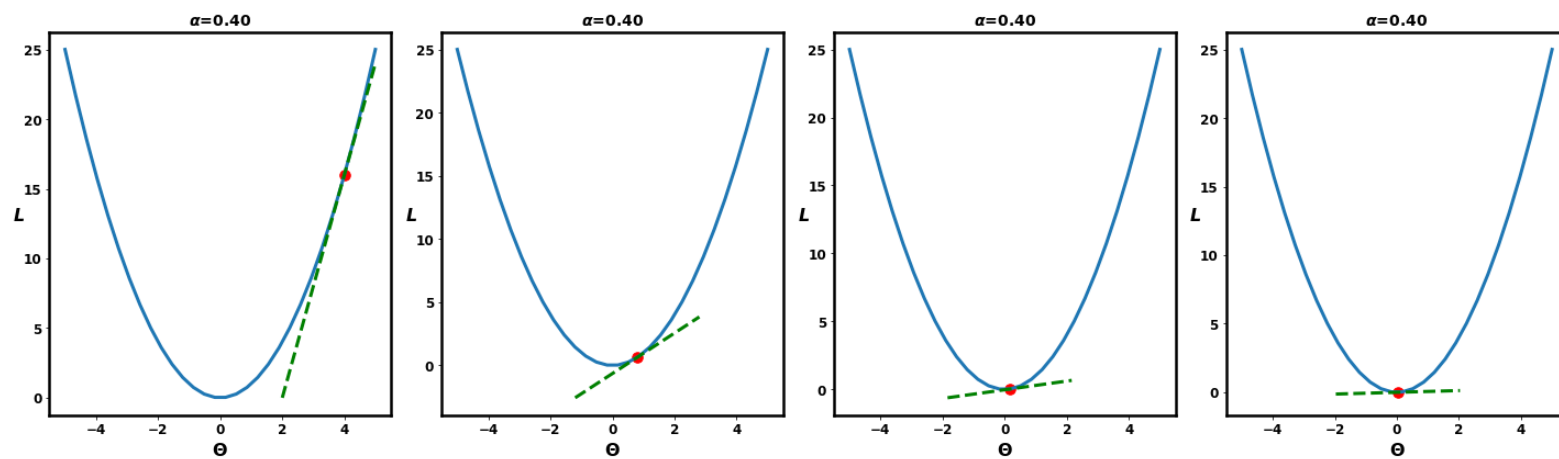


```
In [21]: fig, axs = plot_gradient_descent(alphas= [ alpha ])
```

Below we visualize the progress of the iterative procedure as frames from the movie.

In [22]: fig

Out[22]:



# Gradients: vector derivatives

We illustrated the use of Gradient Descent to find the minimum of a function of a single variable.

The same procedure works when the function is of higher dimension.

Let's illustrate with the MSE Loss often used in Linear Regression, when  $\mathbf{x}^{(i)}$  (and hence  $\Theta$ ) is of dimension  $n$ .

$$\mathbf{y} = \Theta^T \cdot \mathbf{x}$$

With  $(n + 1)$  features (including the constant)

- $\Theta$  is a vector of length  $(n + 1)$
- $\frac{\partial}{\partial \Theta} \mathcal{L}_{\Theta} = \nabla_{\Theta} \mathcal{L}_{\Theta}$ , is a vector of length  $(n + 1)$

$$\nabla_{\Theta} \mathcal{L}_{\Theta} = \begin{pmatrix} \frac{\partial}{\partial \Theta_0} \mathcal{L}_{\Theta} \\ \frac{\partial}{\partial \Theta_1} \mathcal{L}_{\Theta} \\ \vdots \\ \frac{\partial}{\partial \Theta_n} \mathcal{L}_{\Theta} \end{pmatrix}$$

Using MSE Loss as the Loss function

$$\mathcal{L}_{\Theta} = \text{MSE}(\mathbf{y}, \hat{\mathbf{y}}, \Theta) = \frac{1}{m} \sum_{i=1}^m (\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)})^2$$

$$\nabla_{\Theta} \mathcal{L}_{\Theta} = \begin{pmatrix} \frac{\partial}{\partial \Theta_0} \text{MSE}(\mathbf{y}, \hat{\mathbf{y}}, \Theta) \\ \frac{\partial}{\partial \Theta_1} \text{MSE}(\mathbf{y}, \hat{\mathbf{y}}, \Theta) \\ \vdots \\ \frac{\partial}{\partial \Theta_n} \text{MSE}(\mathbf{y}, \hat{\mathbf{y}}, \Theta) \end{pmatrix}$$

Whereas in our code

- We computed derivatives *numerically*
- We will compute them below *analytically*, using calculus

Analytic (closed form) derivatives are much faster to compute.

- During the Deep Learning part of the course, we will see how to *automatically* obtain analytic derivatives

$$\begin{aligned}
\frac{\partial}{\partial \Theta_j} \text{MSE}(\mathbf{y}, \hat{\mathbf{y}}, \Theta) &= \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial \Theta_j} (\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)})^2 && \text{definition} \\
&= \frac{1}{m} \sum_{i=1}^m 2 * (\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}) \frac{\partial}{\partial \Theta_j} (-\hat{\mathbf{y}}^{(i)}) && \text{chain rule} \\
&= -\frac{1}{m} \sum_{i=1}^m 2 * (\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}) \frac{\partial}{\partial \Theta_j} (\Theta * \mathbf{x}^{(i)}) && \hat{\mathbf{y}}^{(i)} = \Theta^T \cdot \mathbf{x}^{(i)} \\
&= -\frac{1}{m} \sum_{i=1}^m 2 * (\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}) \mathbf{x}_j^{(i)} \\
&= -\frac{2}{m} \sum_{i=1}^m (\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}) \mathbf{x}_j^{(i)}
\end{aligned}$$


---



Thus the gradient for Linear Regression can be written in matrix form as

$$\nabla_{\theta} \text{MSE}(X, \theta) = -\frac{2}{m} \mathbf{X}^T (\mathbf{\Theta}^T \mathbf{X} - \mathbf{y}) \quad \text{since } \hat{\mathbf{y}} = \mathbf{\Theta}^T \mathbf{x}$$

This will be particularly useful when working with NumPy as the gradient calculation is a vector operation that is implemented so as to be fast.

# Gradient Descent versus MLE

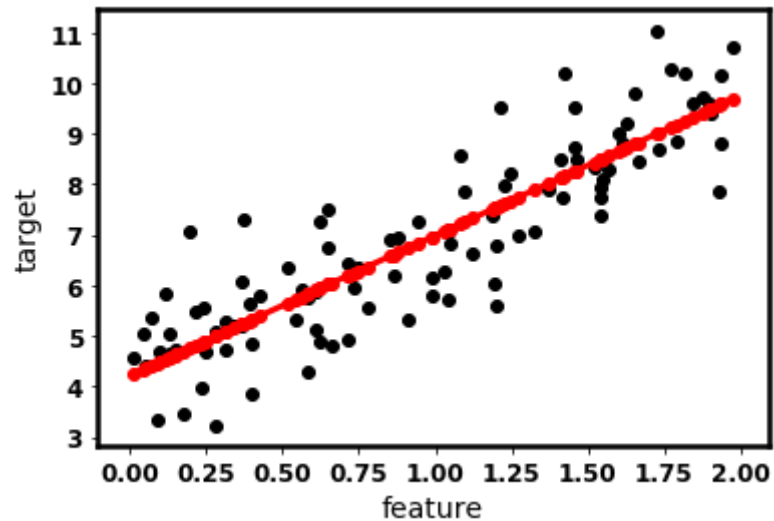
For Linear Regression, there is a closed form solution for finding the optimal  $\Theta$ .

We will demonstrate that the Gradient Descent search comes arbitrarily close.

Let's illustrate Batch Gradient Descent on an example.

First, we use sklearn's `LinearRegression` as a baseline against which we will compare the  $\Theta$  obtained from Gradient Descent.

```
In [23]: X_lr, y_lr = gdh.gen_lr_data()  
         clf_lr = gdh.fit_lr(X_lr, y_lr)  
         fig, ax = gdh.plot_lr(X_lr, y_lr, clf_lr)  
  
         theta_lr = (clf_lr.intercept_, clf_lr.coef_)
```



Now let's perform Batch Gradient Descent and compare the  $\Theta$ 's

```
In [24]: gd_theta = gdh.batchGradientDescent_lr(X_lr, y_lr)
         theta_lr = gd_theta
```

```
Out[24]: array([[7.993605777301127e-15],
                [-8.43769498715119e-15]], dtype=object)
```

The  $\Theta$ 's are equal up to 15 decimal points.

Let's look at the code for Batch Gradient Descent and examine the details



```
alpha = 0.1 n_iterations = 1000 m = 100 theta = np.random.randn(2,1)
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - alpha * gradients
```

- We use the closed form, analytic expression for the gradient
- We update

$$\Theta = \Theta - \alpha * \text{gradient}$$

Notice that the "step size" ( $\alpha * \text{gradient}$ )

- Is "big" when the gradient is large
- Is "small" when the gradient is small (close to optimal)

Since the  $\Theta$ 's computed by Gradient Descent and Linear Regression are the same, it's no surprise that the predictions are too.

- As demonstrated in the following code

```
In [25]: X_new = np.array([[0], [2]])
gd_y_pred = gdh.predict(X_new, theta_lr)
clf_y_pred = clf_lr.predict(X_new)

gd_y_pred == clf_y_pred
```

```
Out[25]: array([[ True],
               [ True]])
```

# Gradient Descent in depth

There are many subtleties to Gradient Descent.

As Gradient Descent will be a *key tool* in the Deep Learning part of the course, we briefly explore a few issues below.

# How big should $\alpha$ be ?

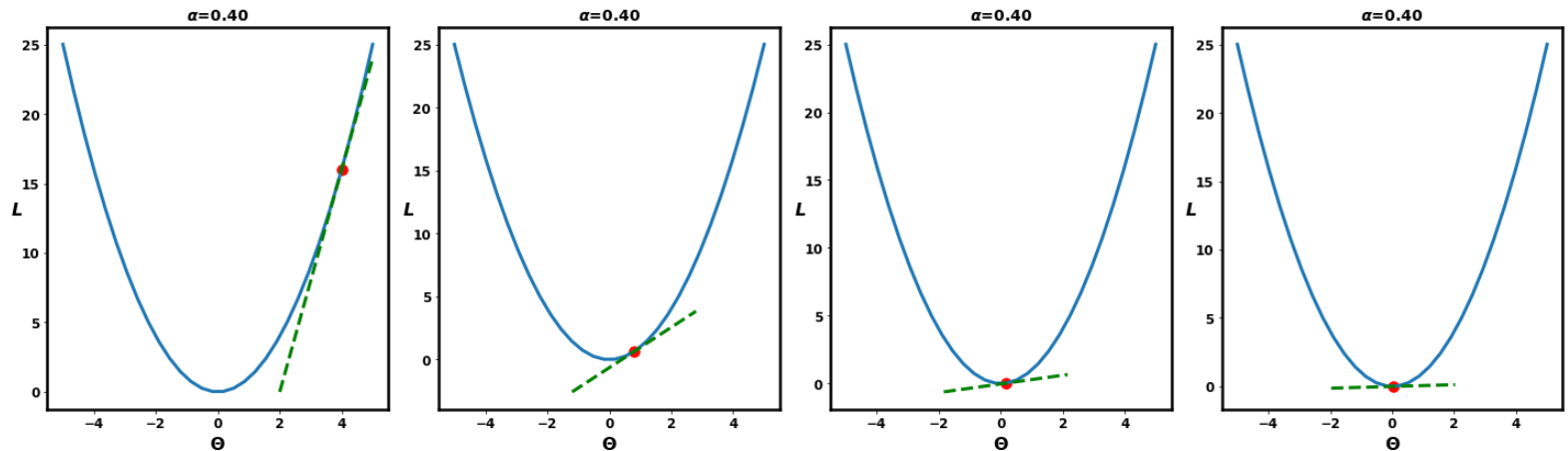
The "step size" we take along the direction of the gradient is  $\alpha$ .

Does the choice of  $\alpha$  matter ?

Here are 4 steps with  $\alpha = 0.40$

```
In [26]: fig, axs = plot_gradient_descent(alphas= [ alpha ])
fig
```

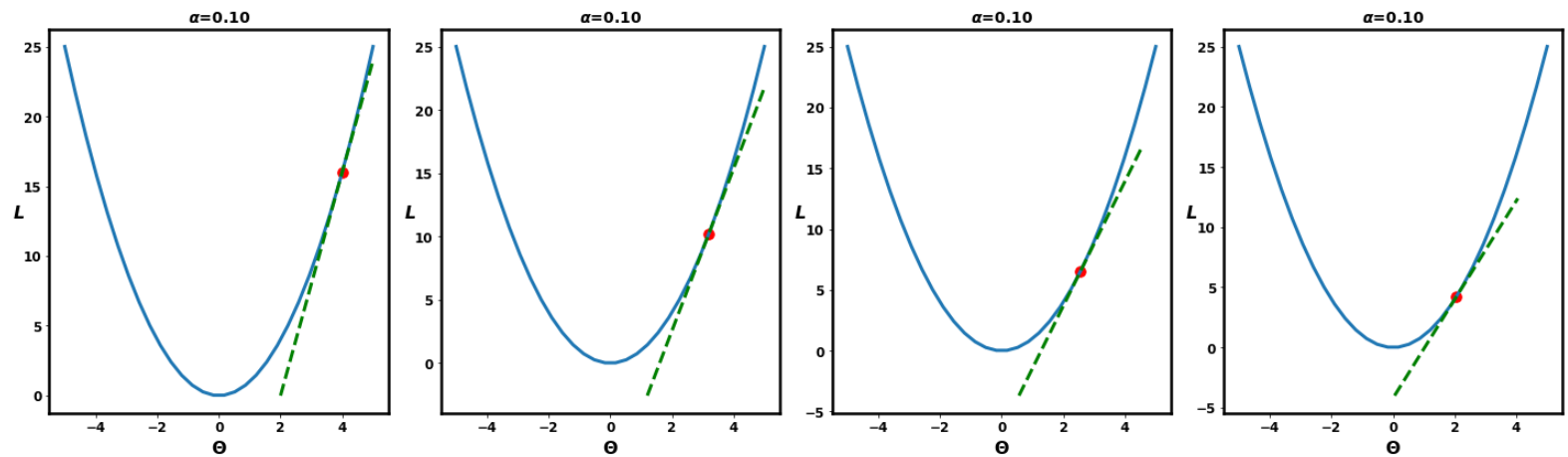
Out[26]:



And with a much smaller  $\alpha = 0.1$

```
In [27]: fig, axs = plot_gradient_descent(alphas= [ 0.1 ])
fig
```

Out[27]:



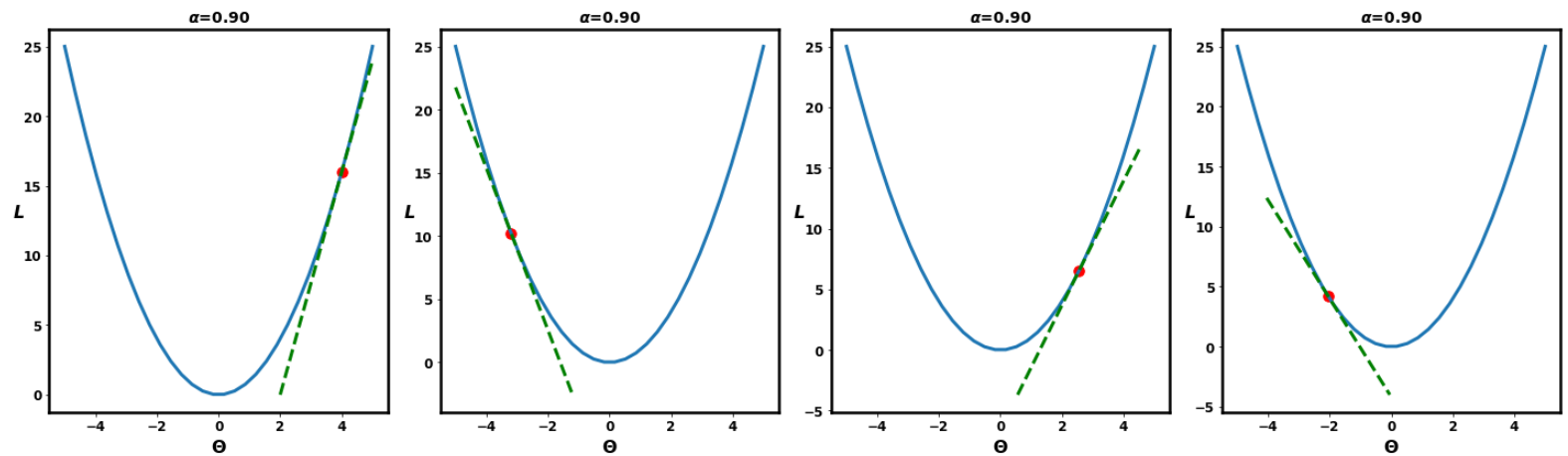


Convergence toward the optimal is much slower.

What if we used a larger  $\alpha = 0.9$  ?

```
In [28]: fig, axs = plot_gradient_descent(alphas= [ 0.9 ])
fig
```

Out[28]:



You can see that we over-shoot the optimal repeatedly.

This may be problematic

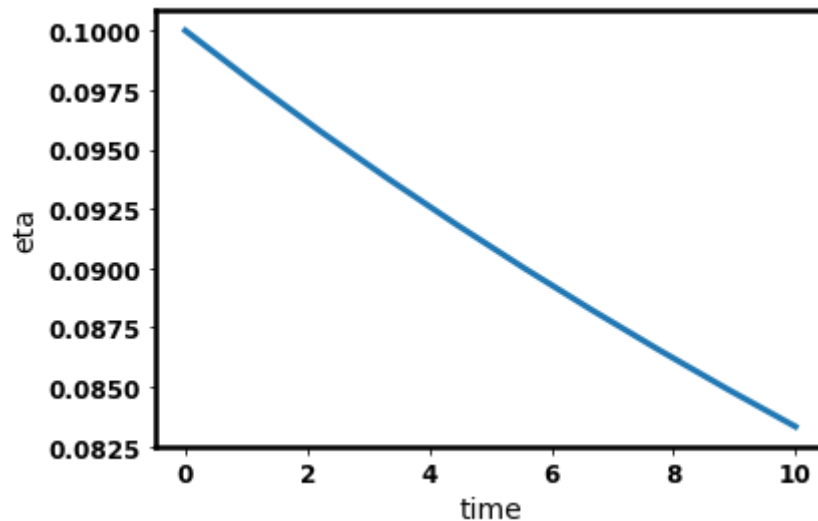
- For more complex loss functions: we may "skip" over a local optimum

An adaptive learning rate schedule may be the solution:

- Take big steps at first
- Take smaller steps toward end

In [29]: `t0, t1 = 5, 50 # learning schedule hyperparameters`

```
def learning_schedule(t):  
    return t0 / (t + t1)  
  
t = np.linspace(0, 10, 10)  
  
fig = plt.figure()  
ax = fig.add_subplot(1,1,1)  
_ = ax.plot(t, learning_schedule(t))  
_ = ax.set_xlabel("time")  
_ = ax.set_ylabel("eta")
```



We will revisit Learning Rate schedules in the Deep Learning part of the course.

## Mini batch Gradient Descent

The Average Loss function in Classical Machine Learning has the form

$$\mathcal{L}_{\Theta} = \frac{1}{m} \sum_{i=1}^m \mathcal{L}_{\Theta}^{(i)}$$

That is, it is composed of  $m$  sub-expressions where  $m$  is the number of training examples.

- Each subexpression requires a computation and a derivative

Thus, for large sets of training examples, Gradient Descent can be expensive.

It may be possible to *approximate*  $\mathcal{L}_\Theta$  using fewer than  $m$  expressions.

- Choose a *random subset* ( of size  $m' \leq m$ ) of examples:  $I = \{i_1, \dots, i_{m'}\}$
- Approximate  $\mathcal{L}_\Theta$  on  $I$

$$\mathcal{L}_\Theta \approx \frac{1}{|I|} \sum_{i \in I} \mathcal{L}_\Theta^{(i)}$$



This means it is possible to update  $\Theta$  after evaluating only  $m' \leq m$  expressions.

Whereas Gradient Descent computes an exact  $\mathcal{L}_{\Theta}$  to perform a single update of  $\Theta$ :

*Mini batch Gradient Descent*

- Takes  $b = m/m'$  smaller steps, each updating  $\Theta$
- Each small step using an approximation of  $\mathcal{L}_{\Theta}$  based on  $m' \leq m$  examples

It does this by

- Choosing batch size  $m'$
- Partitioning the set of example indices  $\{i | 1 \leq i \leq m\}$ 
  - into  $b$  batches of size  $m'$
  - Batch  $i' : b_{(i')}$  is one partition consisting of  $m'$  example indices
  - Each small step uses a single batch to approximate  $\mathcal{L}_{\Theta}$  and update  $\Theta$

The collection of  $b$  small steps (comprising all examples) is called an *epoch*

So one epoch of Mini batch Gradient Descent performs  $b$  updates.

When batch size  $m' = m$ , we have our original algorithm known as *Batch Gradient Descent*.

How does one choose  $m' \leq m$ ?

- Want  $m'$  large enough so approximations aren't too noisy
  - Don't want losses of the mini-batches of each epoch to be too different
- Often determined by *external* considerations
  - GPU memory (preview of Deep Learning)

# Initializing $\Theta$

As we will see in the Deep Learning part of the course

- Initial  $\Theta$  is *not a trivial* choice

Consider a Loss function like the Hinge Loss

- Our initial choice of  $\Theta$  could leave us in a *flat* area of the Loss function
- No derivative, but maybe not optimal
- No way to escape !

## When to stop

Deciding when to stop the iterative process is another choice to be made

- Stop when decrease in  $\mathcal{L}_\Theta$  is "too small"

# Improvements to Gradient Descent

Simon Ruder survey (<https://arxiv.org/abs/1609.04747>).

Gradient Descent Cheatsheet (<https://towardsdatascience.com/10-gradient-descent-optimisation-algorithms-86989510b5e9>).

The update step

$$\Theta = \Theta - \alpha * \frac{\partial \mathcal{L}_{\Theta}}{\partial \Theta}$$

where  $\alpha$  is the learning rate.

The improvements to Gradient Descent modify

- $\alpha$ , the learning rate
- $\frac{\partial \mathcal{L}_{\Theta}}{\partial \Theta}$  the gradient



In order to be able to flexibly change the definition of both the gradient and the learning rate at each time step  $t$ , we will re-write the update step at time  $t$  as

$$\Theta_{(t)} = \Theta_{(t-1)} - \alpha' * V_{(t)}$$

$V_{(t)}$  will be our modified gradient and  $\alpha'$  our modified learning rate.

## Momentum: modify the gradient

In vanilla Gradient Descent, the gradients at time  $(t - 1)$  and time  $t$  are completely independent.

This has the potential for gradients to rapidly change direction (recall, they are a vector).

To smooth out jumps we could compute a modified gradient  $V_{(t)}$  as:

$$V_{(t)} = \beta_V * V_{(t-1)} + (1 - \beta_V) * \frac{\partial \mathcal{L}_\Theta}{\partial \Theta}$$

(Initialize  $V_0 = 0$ )

That is, the modified gradient is a weighted combination of the previous gradient and the new gradient.

Typically  $\beta_V \approx 0.9$  so the old gradient dominates.

$V_{(t)}$  is the exponentially weighted moving average of the gradient.

Hence, there is "momentum" in the gradients in that they can't jump suddenly.

## RMSprop: Modify the learning rate

Let

$$S_{(t)} = \beta_S * S_{(t-1)} + (1 - \beta_S) * \left( \frac{\partial \mathcal{L}_\Theta}{\partial \Theta} \right)^2$$

That is,  $S_{(t)}$  is the exponentially weighted *variance* of the gradient.

(Initialize  $S_0 = 0$ )

Rather than using a learning rate of  $\alpha$ , the RMSprop algorithm uses

$$\alpha' = \frac{1}{\sqrt{S_{(t)} + \epsilon}} * \alpha$$

The intuition is that if the gradient with respect to  $\Theta_j$  is noisy (i.e., large variance) we want to damp updates in that component.

This also has the advantage that

- A rarely updated element  $\Theta_i$ , having a low variance,
- Will have a relatively larger update when it is encountered than a more frequently encountered feature.

Typically  $\beta_S \approx 0.9$  so the old variance dominates.

Why the extra  $\epsilon$ ? We've seen this before (e.g.,  $\log(x + \epsilon)$ ): it's to avoid mathematical issues of certain functions (inverse, log) when the argument is 0.

## AdaM: Modify both the gradient and the learning rate

The AdaM (Adaptive Moment) algorithm modifies *both*

- The gradient
- The learning rate

via exponentially weighted moving averages of the gradient as well as its variance.

$$\begin{aligned}V_{(t)} &= \beta_V * V_{(t-1)} + (1 - \beta_V) \frac{\partial \mathcal{L}_\Theta}{\partial \Theta} \\S_{(t)} &= \beta_S * S_{(t-1)} + (1 - \beta_S) * \left(\frac{\partial \mathcal{L}}{\partial \Theta}\right)^2 \\ \alpha' &= \frac{1}{\sqrt{S_{(t)} + \epsilon}} * \alpha\end{aligned}$$

## Bias correction

You will have observed that we initialized to 0 the moving averages for gradients ( $V_0 = 0$ ) and the variance of the gradients ( $S_0 = 0$ ).

So the values are "biased" towards 0 with the bias having greatest effect for small  $t$  (i.e., when the number of "actual" values is small).

We can correct for the bias by dividing by  $(1 - \beta^t)$ :

$$\begin{aligned}\hat{V} &= \frac{V_{(t)}}{1 - \beta_V^t} \\ \hat{S} &= \frac{S_{(t)}}{1 - \beta_S^t}\end{aligned}$$



In [30]: `print("Done")`

Done