Machine Learning

|Gradient Descent

Prof. Dr. Mostafa Elhosseini

https://youtube.com/@ProfElhosseiniTechVerse

Agenda

- Iterative optimization approach
  - Gradient Descent
  - Batch
  - Stochastic
  - Minibatch

Intro…

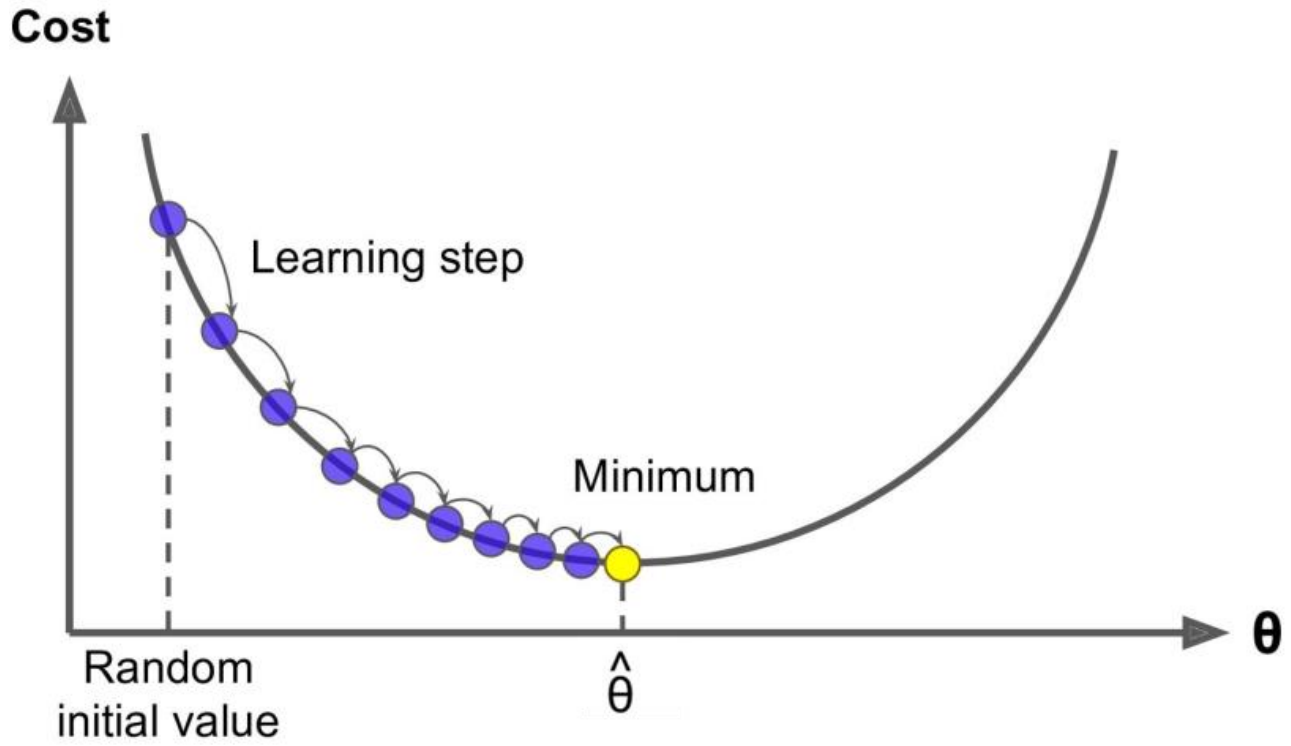
- We will look at very different ways to train a Linear Regression model, better suited for cases where there are a large number of features, or too many training instances to fit in memory

Gradient Descent

- *Gradient Descent (النزول التدريجي)* is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems.
- The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function
  - Suppose you are lost in the mountains in a dense fog; you can only feel the slope of the ground below your feet.
  - A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope (الأشد انحدارا).
  - This is exactly what Gradient Descent does:
    - → it measures the local gradient of the error function with regards to the parameter vector θ, and it goes in the direction of descending gradient.
    - → Once the gradient is zero, you have reached a minimum

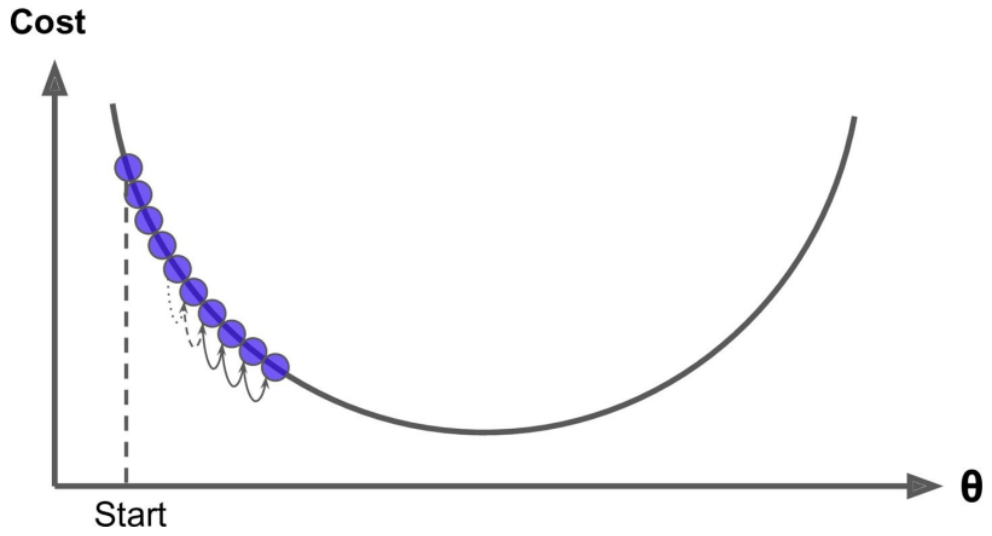
Gradient Descent

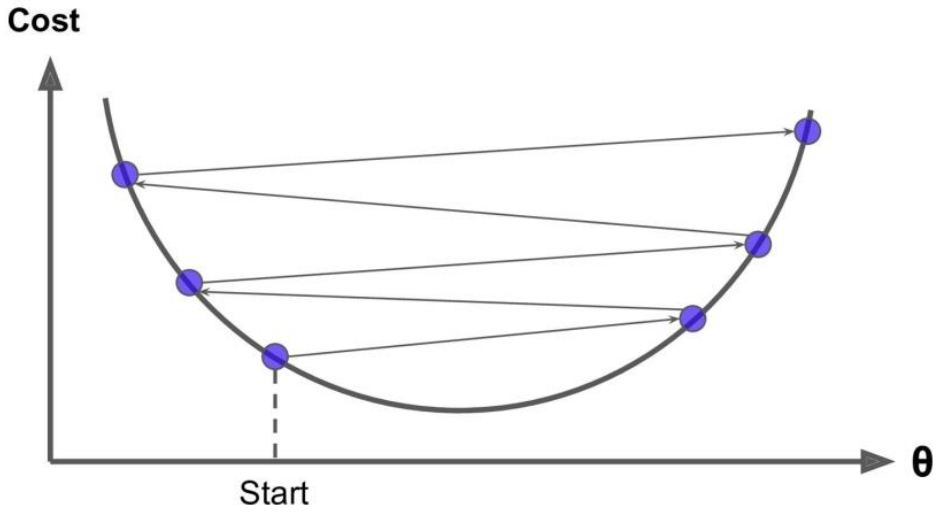
- you start by filling θ with random values
  - this is called *random initialization*, and
- then you improve it gradually, taking one baby step at a time,
  - each step attempting to decrease the cost function (e.g., the MSE),
  - until the algorithm converges to a minimum



Gradient Descent

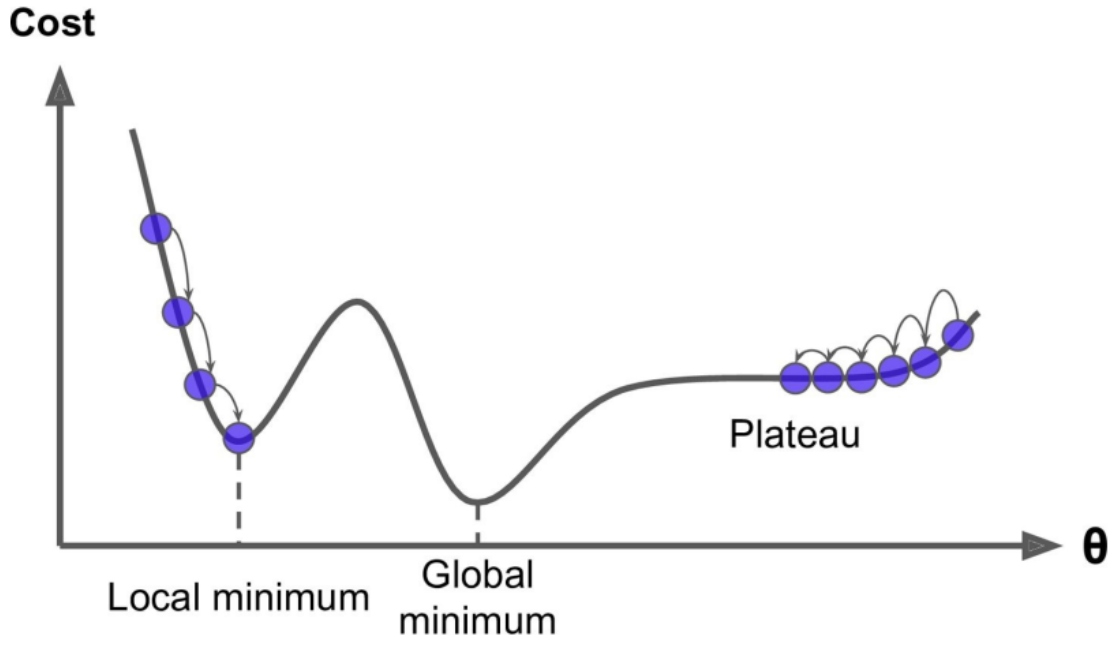
- An important parameter in Gradient Descent is the size of the steps, determined by the *learning rate* hyperparameter.
  - If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time
  - if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution





Cost function shapes

- not all cost functions look like nice regular bowls.
  - There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum very difficult
  - if the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*
  - If it starts on the right, then it will take a very long time to cross the plateau, and if you stop too early you will never reach the global minimum



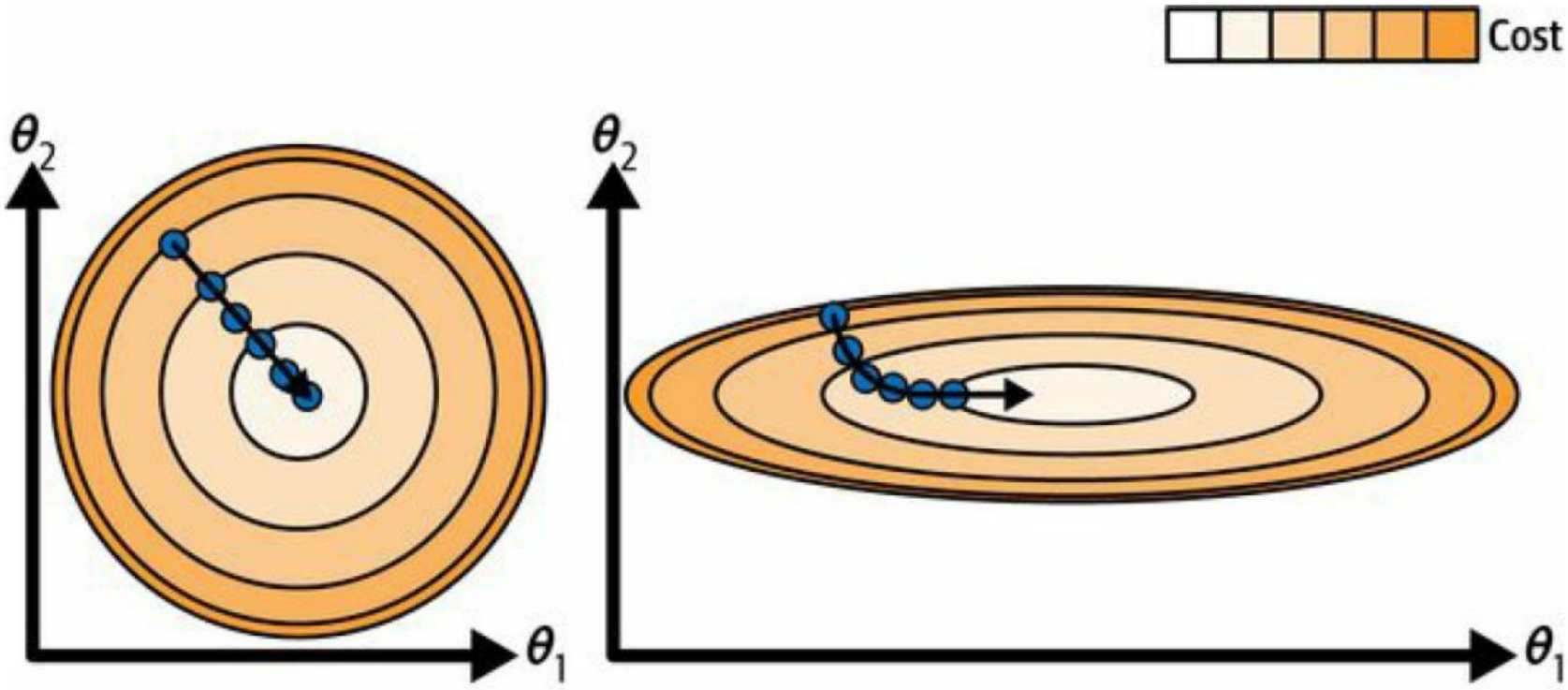
Cost function shapes

- Fortunately, the MSE cost function for a Linear Regression model happens to be a *convex function (دالة محدبة)*,
  - if you pick any two points on the curve, the line segment joining them never crosses the curve.
  - This implies that there are no local minima, just one global minimum.
  - It is also a continuous function with a slope that never changes abruptly

These two facts have a great consequence:

- Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).
- In fact, the cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales

Feature scaling effect



When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's StandardScaler class), or else it will take much longer to converge.

Note that:

- Training a model means searching for a combination of model parameters that minimizes a cost function (over the training set).
  - It is a search in the model's *parameter space*
  - The more parameters a model has, the more dimensions this space has, and the harder the search is
  - Searching for a needle in a 300-dimensional haystack is much trickier than in three dimensions
  - Fortunately, since the cost function is convex in the case of Linear Regression, the needle is simply at the bottom of the bowl

Batch Gradient Descent

- To implement Gradient Descent, you need to compute the gradient of the cost function with regards to each model parameter θ_j
  - Partial derivative
  - It is like asking "what is the slope of the mountain under my feet if I face east?
  - Instead of computing these gradients individually, you can use gradient vector to compute them all in one go - Nabla.

$$\nabla_{\theta} MSE(\theta) = \begin{bmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{bmatrix}$$

Batch Gradient Descent

- Notice that $\boldsymbol{\nabla_\theta}\, MSE(\theta)$ involves calculations over the full training set X, at each Gradient Descent step!
  - This is why the algorithm is called *Batch Gradient Descent*:
  - It uses the whole batch of training data at every step
  - it is terribly slow on very large training sets
  - Gradient Descent scales well with the number of features
  - training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation
- Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\boldsymbol{\nabla_\theta}\, MSE(\theta)$ from θ

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta \, \text{MSE}(\theta)$$

Batch Gradient Descent

- η: learning rate controls the size of the downhill steps
- **Epoch**: Each iteration over the training set

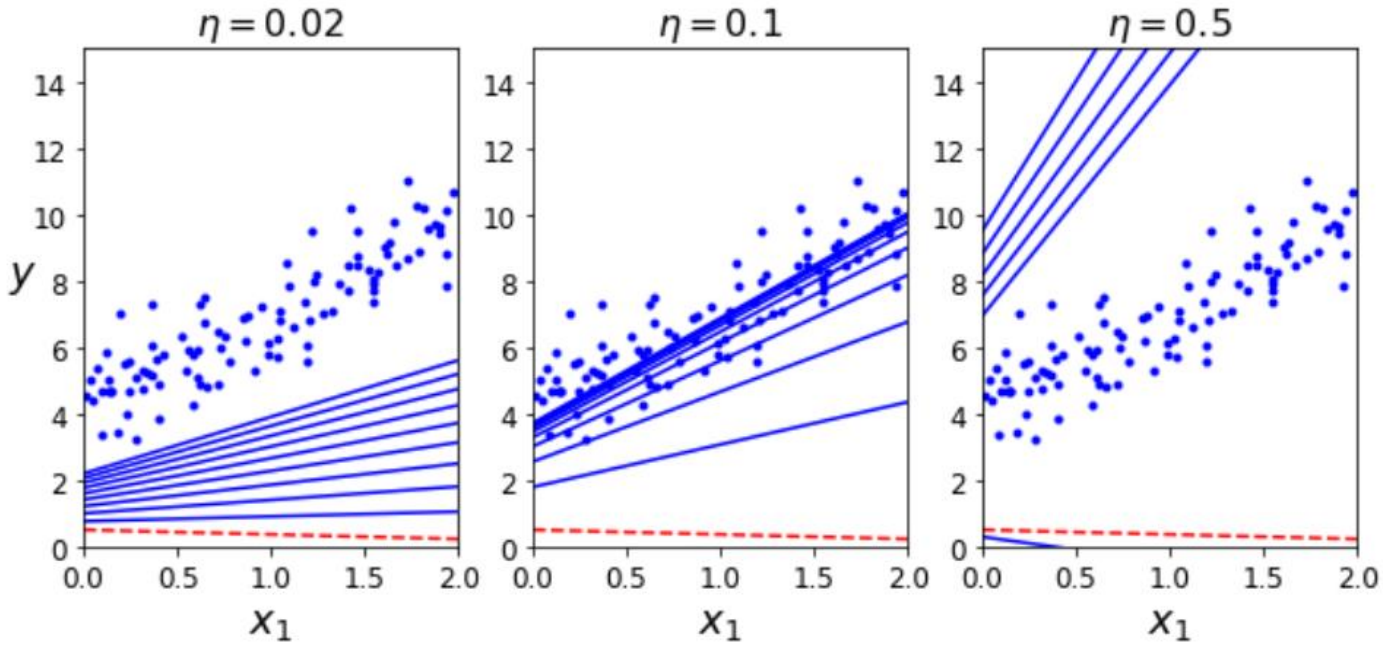
```
eta = 0.1  # learning rate
n_epochs = 1000
m = len(X_b)  # number of instances

np.random.seed(42)
theta = np.random.randn(2, 1)  # randomly initialized model parameters

for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients
```

```
theta
```

```
array([[4.21509616],
       [2.77011339]])
```

learning rate

- η: learning rate controls the size of the downhill steps
  - too low learning rate: the algorithm will eventually reach the solution, but it will take a long time
  - Not too high, not too low learning rate: in just a few epochs, it has already converged to the solution.
  - Too high learning rate: the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step
  - To find a good learning rate, you can use **grid search**



Stopping criteria

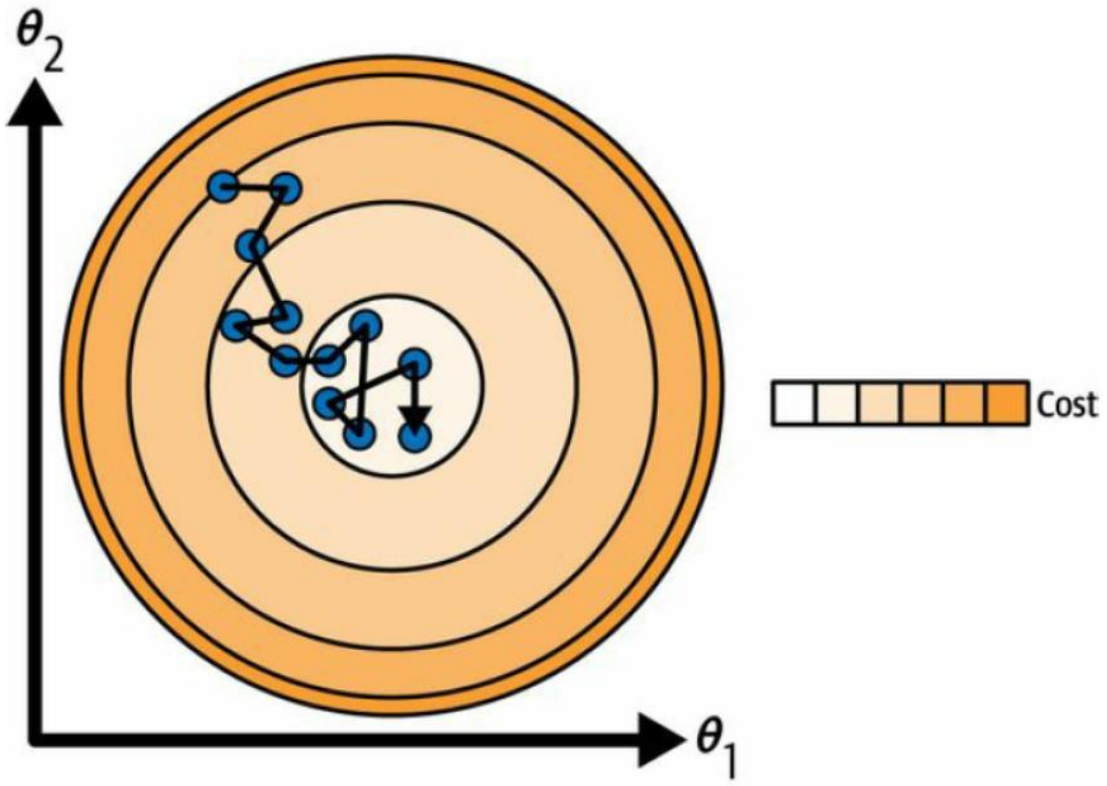
- You may wonder how to set the number of epochs.
  - If it is too low, you will still be far away from the optimal solution when the algorithm stops, but
  - If it is too high, you will waste time while the model parameters do not change anymore.
- A simple solution is to set a very large number of epochs but to interrupt the algorithm when the gradient vector becomes tiny —
  - when the norm becomes smaller than *tolerance* ϵ

Stochastic Gradient Descent

- The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large
- *Stochastic Gradient Descent* just picks a random instance in the training set at every step and computes the gradients based only on that single instance.
  - This makes the algorithm much faster since it has very little data to manipulate at every iteration.
  - It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration

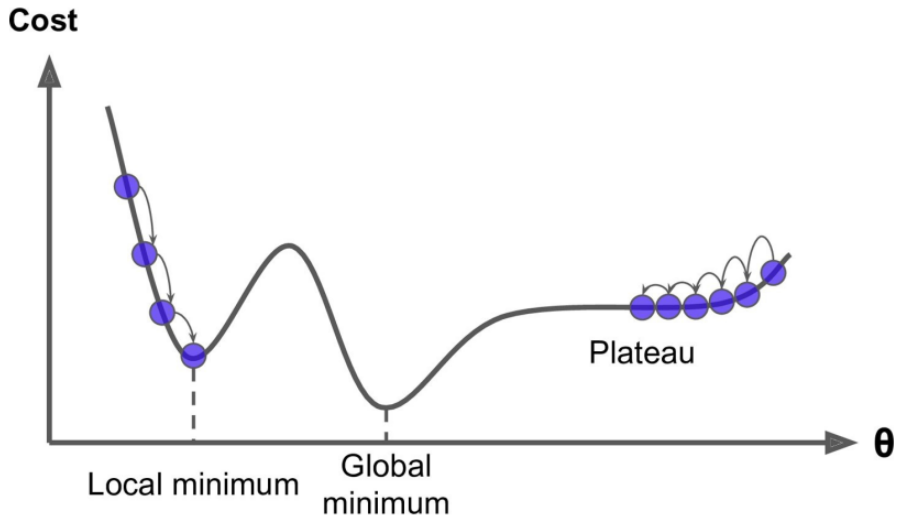
Stochastic Gradient Descent

- Due to its stochastic nature, this algorithm is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average.
- Once the algorithm stops, the final parameter values are good, but not optimal



Stochastic Gradient Descent

- When the cost function is very irregular, this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does
  - Randomness is good to escape from local optima
  - but bad because it means that the algorithm can never settle at the minimum
  - One solution to this dilemma is to gradually reduce the learning rate ↓

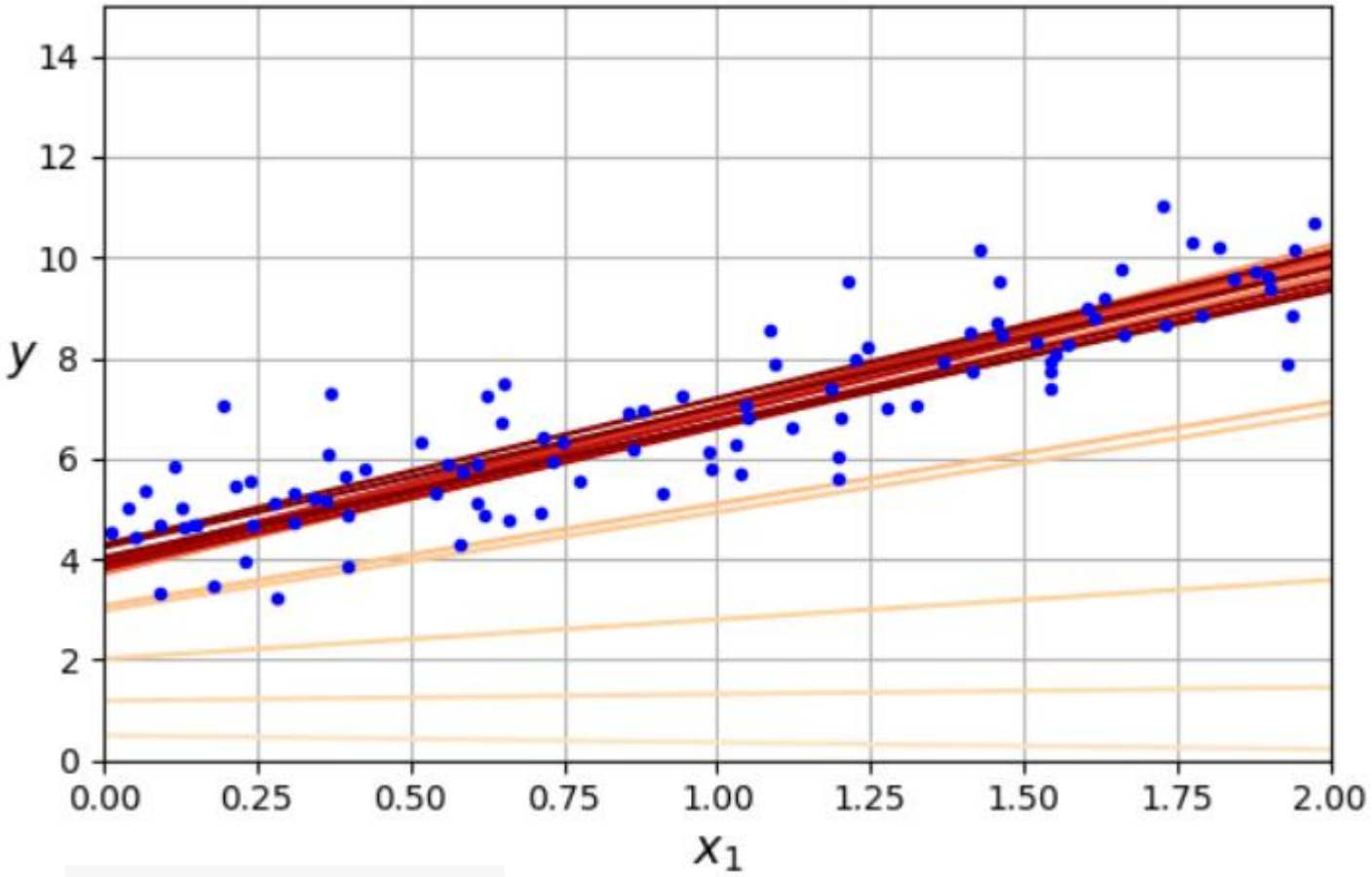


Stochastic Gradient Descent

- One solution to this dilemma is to gradually reduce the learning rate.
  - The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum.
  - This process is called *simulated annealing*, because it resembles the process of annealing in metallurgy where molten metal is slowly cooled down.
  - The function that determines the learning rate at each iteration is called the *learning schedule*.
  - If the learning rate is reduced too quickly, you may get stuck in a local minimum, or even end up frozen halfway to the minimum

Stochastic Gradient Descent

```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

np.random.seed(42)
theta = np.random.randn(2, 1)  # random initialization

n_shown = 20  # extra code – just needed to generate the figure below
plt.figure(figsize=(6, 4))  # extra code – not needed, just formatting

for epoch in range(n_epochs):
    for iteration in range(m):

        # extra code – these 4 lines are used to generate the figure
        if epoch == 0 and iteration < n_shown:
            y_predict = X_new_b @ theta
            color = mpl.colors.rgb2hex(plt.cm.OrRd(iteration / n_shown + 0.15))
            plt.plot(X_new, y_predict, color=color)

        random_index = np.random.randint(m)
        xi = X_b[random_index : random_index + 1]
        yi = y[random_index : random_index + 1]
        gradients = 2 * xi.T @ (xi @ theta - yi)  # for SGD, do not divide by m
        eta = learning_schedule(epoch * m + iteration)
        theta = theta - eta * gradients
        theta_path_sgd.append(theta)  # extra code – to generate the figure

# extra code – this section beautifies and saves Figure 4–10
plt.plot(X, y, "b.")
plt.xlabel("$x_1$")
plt.ylabel("$y$", rotation=0)
plt.axis([0, 2, 0, 15])
plt.grid()
save_fig("sgd_plot")
plt.show()
```




```
theta
```

```
array([[4.21076011],
       [2.74856079]])
```

Stochastic Gradient Descent

- By convention we iterate by rounds of m iterations; each round is called an epoch
- since instances are picked randomly, some instances may be picked several times per epoch while others may not be picked at all
  - shuffle the training set, then go through it instance by instance, then shuffle it again, and so on. However, this generally converges more slowly

Stochastic GD using Scikit-Learn

- Use the SGDRegressor class, which defaults to optimizing the squared error cost function.
- The following code runs 50 epochs, starting with a learning rate of 0.1 ($\eta_0 = 0.1$), using the default learning schedule (different from the preceding one), and it does not use any regularization (penalty=None; more details on this shortly):

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-5, penalty=None, eta0=0.01,
                       n_iter_no_change=100, random_state=42)
sgd_reg.fit(X, y.ravel())  # y.ravel() because fit() expects 1D targets
```

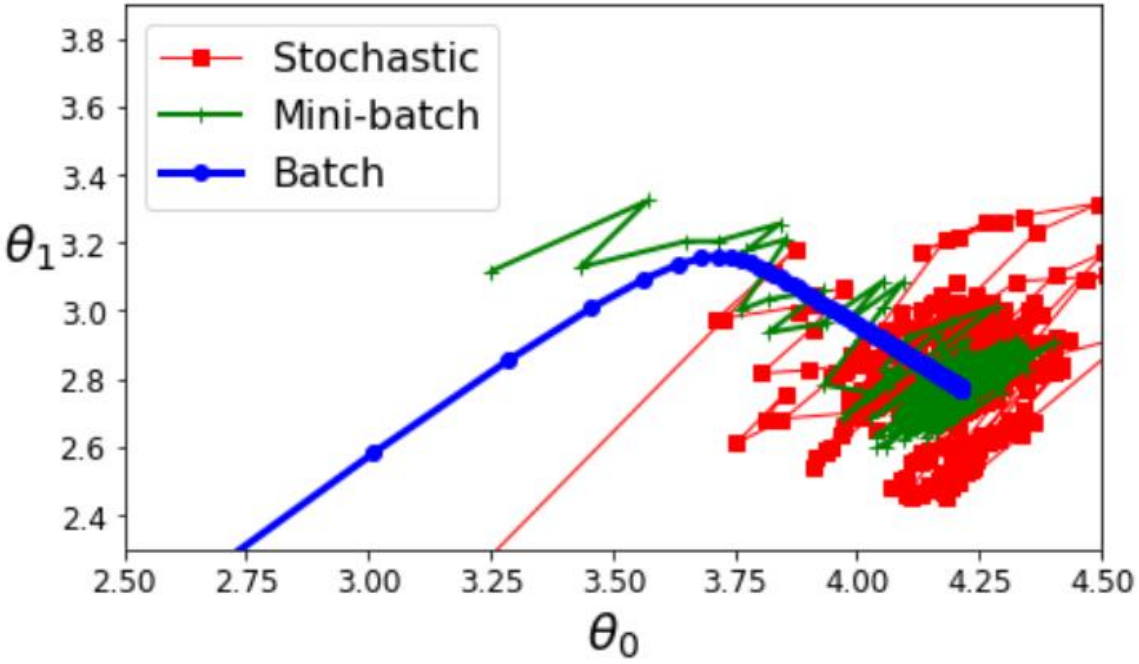
```
SGDRegressor
SGDRegressor(n_iter_no_change=100, penalty=None, random_state=42, tol=1e-05)
```

```
sgd_reg.intercept_, sgd_reg.coef_
```

```
(array([4.21278812]), array([2.77270267]))
```

Mini-batch Gradient Descent

- Instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-batch GD computes the gradients on small random sets of instances called *minibatches*
- The main advantage of mini-batch GD over stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs
- The algorithm's progress in parameter space is less erratic than with SGD, especially with fairly large mini-batches. As a result, Mini-batch GD will end up walking around a bit closer to the minimum than SGD. But, on the other hand, it may be harder for it to escape from local minima



Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	n/a
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	n/a

Next Lecture

- Polynomial regression