Hands-on Machine Learning

4. Training Linear Models

How Things Work

- Having a good understanding of how things work can help you:
  - choose the appropriate model.
  - choose the right training algorithm to use.
  - choose a good set of hyperparameters for your task.
  - debug issues and perform error analysis more efficiently.

- What you learn about linear models will be essential in understanding, building, and training neural networks.

1.
Linear Regression

Linear Model

- A linear model makes a prediction by computing a weighted sum of the input features, plus a constant called the *bias term* (also called the *intercept term*):

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_i is the $i\text{-}th$ feature value.
- θ_j is the $j\text{-}th$ model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \ldots, \theta_n$).

Matrix Form of Linear Model

- Linear model can be written in matrix form using vectors:

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}.\mathbf{x}$$

- $\boldsymbol{\theta}$ is the model's *parameter vector*: $\boldsymbol{\theta} = [\theta_0 \ \theta_1 \ \theta_2 \ \dots \ \theta_n]$
- $\mathbf{x}$ is the instance's *feature vector*: $\mathbf{x} = [x_0 \ x_1 \ x_2 \ \dots \ x_n]$ with $x_0 = 1$.
- $\boldsymbol{\theta}.\mathbf{x}$ is the dot product of the vectors $\boldsymbol{\theta}$ and $\mathbf{x}$:

$$\boldsymbol{\theta}.\mathbf{x} = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- $h_{\boldsymbol{\theta}}$ is the hypothesis function, using the model parameters $\boldsymbol{\theta}$.
- If $\boldsymbol{\theta}$ and $\mathbf{x}$ are column vectors, then the prediction is $\hat{y} = \boldsymbol{\theta}^T \mathbf{x}$ where $\boldsymbol{\theta}^T$ is the *transpose* of $\boldsymbol{\theta}$.

Training the Model

- To train a Linear Regression model, we need to find the value of $\boldsymbol{\theta}$ that minimizes the Mean Squared Error (MSE).

$$\mathrm{MSE}\left(\mathbf{X}, h_{\boldsymbol{\theta}}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\mathrm{T}}\mathbf{x}^{(i)} - y^{(i)}\right)^2$$

- To find the value of $\boldsymbol{\theta}$ that minimizes the cost function, there is a closed-form solution, which is called the *Normal Equation*:

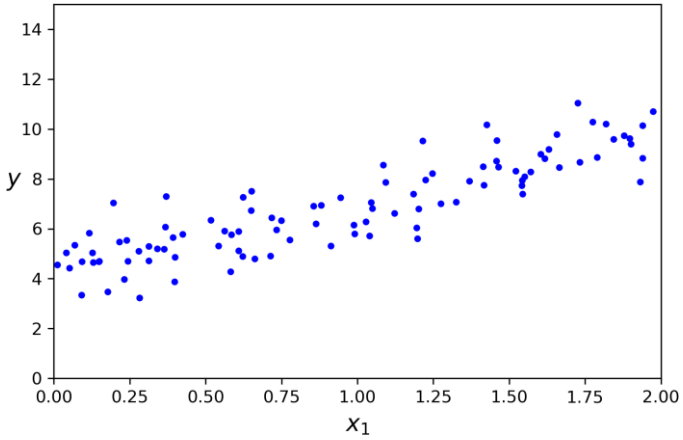
$$\widehat{\boldsymbol{\theta}} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

- $\widehat{\boldsymbol{\theta}}$ is the value of $\boldsymbol{\theta}$ that minimizes the cost function.
- $\mathbf{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

Example

- Generate some linear-looking data:

```
import numpy as np

np.random.seed(42)  # to make this code example reproducible
m = 100  # number of instances
X = 2 * np.random.rand(m, 1)  # column vector
y = 4 + 3 * X + np.random.randn(m, 1)  # column vector
```



- Compute $\widehat{\boldsymbol{\theta}}$ using the Normal Equation:

```
from sklearn.preprocessing import add_dummy_feature

X_b = add_dummy_feature(X)  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```

```
theta_best
```

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

Comparing with `LinearRegression`

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
```

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

- The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function.
- This function computes $\widehat{\boldsymbol{\theta}} = \mathbf{X}^{+}\mathbf{y}$ where $\mathbf{X}^{+}$ is the pseudoinverse of $\mathbf{X}$.
- The complexity of inverting $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is of $O(n^{2.376})$ to $O(n^{3})$.
- The complexity of computing pseudoinverse using Singular Value Decomposition (SVD) is $O(mn + n^{2})$.

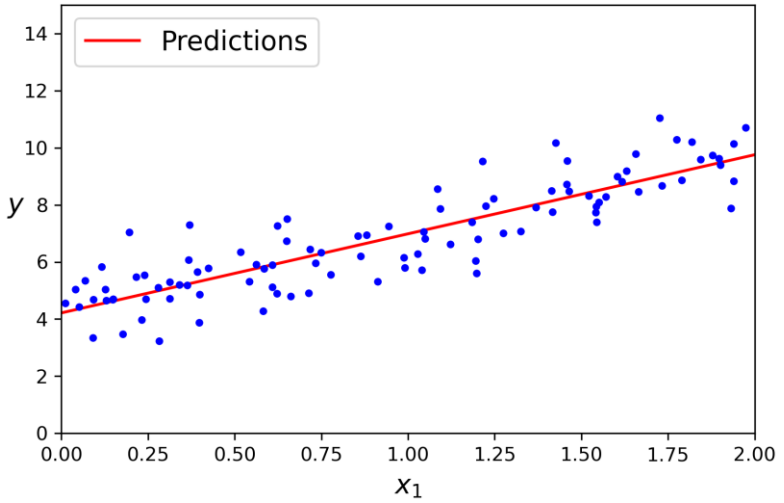
Make a Prediction

```
X_new = np.array([[0], [2]])
X_new_b = add_dummy_feature(X_new)  # add x0 = 1 to each instance
y_predict = X_new_b @ theta_best
y_predict
```

```
array([[4.21509616],
       [9.75532293]])
```

```
lin_reg.predict(X_new)
```

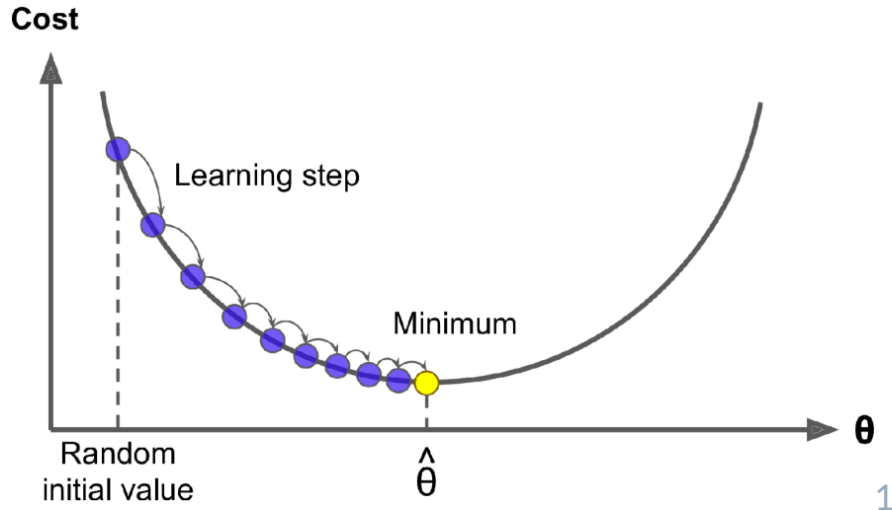
```
array([[4.21509616],
       [9.75532293]])
```



2.
Gradient Descent

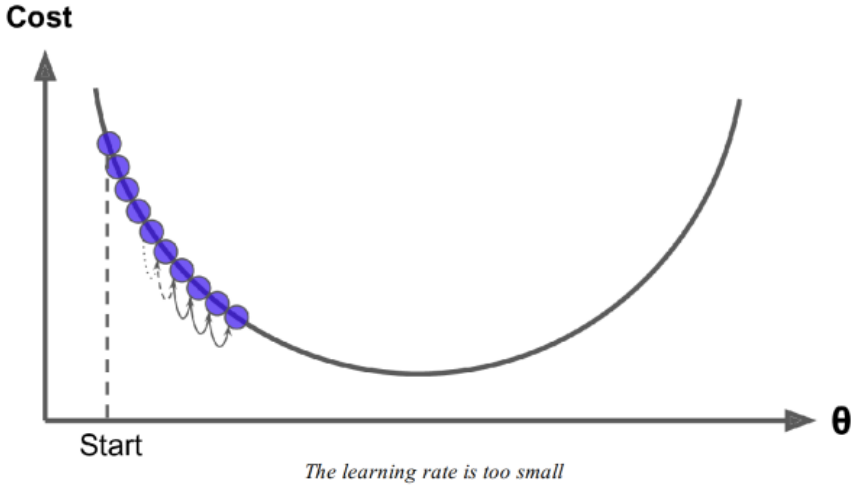
Gradient Descent

- *Gradient Descent* 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.
- We start by filling **θ** with random values (*random initialization*).
- 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.

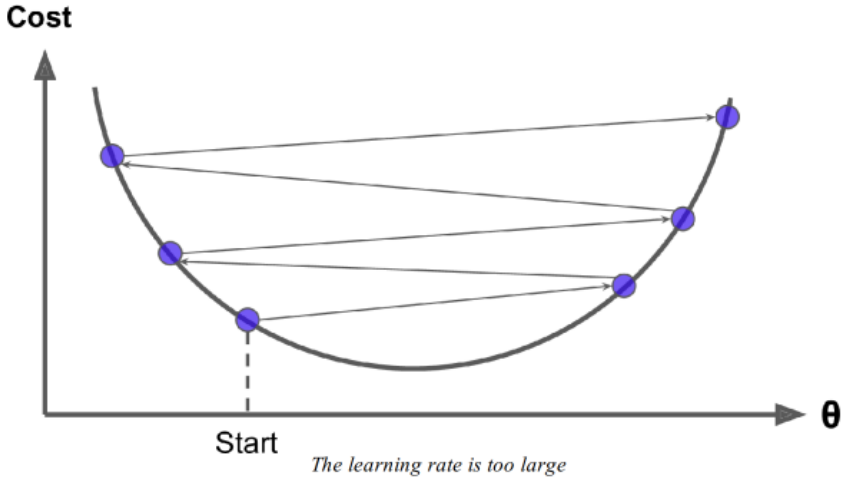


Learning Rate

- *Learning Rate (η):* a hyperparameter of gradient descent that determines the size of the steps.
  - Too small: convergence will take a long time.
  - Too large: the algorithm might diverge.



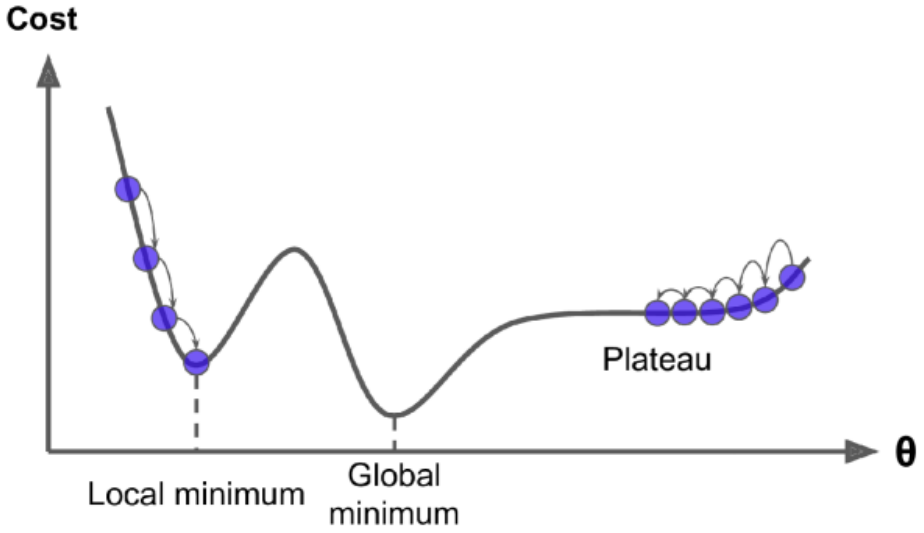
The learning rate is too small



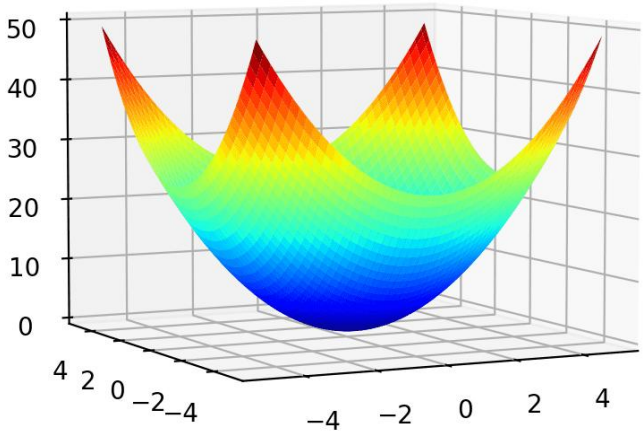
The learning rate is too large

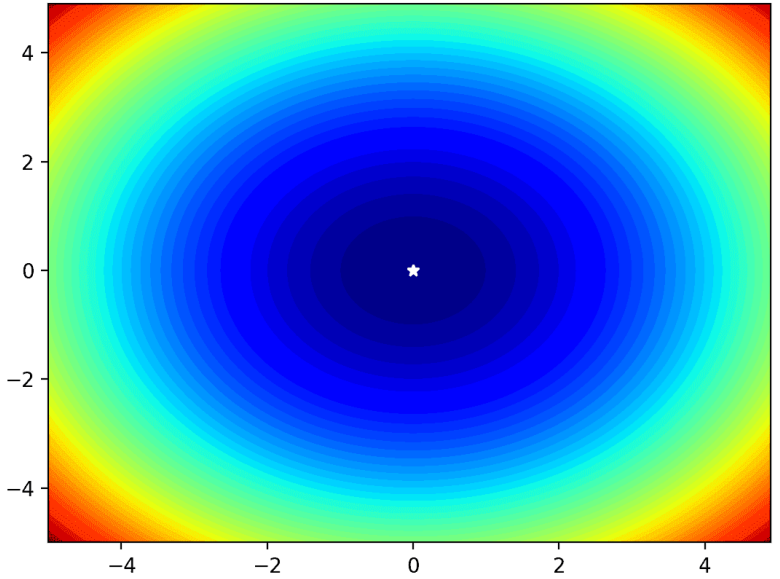
Challenges with Gradient Descent

- The MSE cost function for a Linear Regression model is a *convex function*.
- MSE is a continuous function with a slope that never changes abruptly.
  - 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).



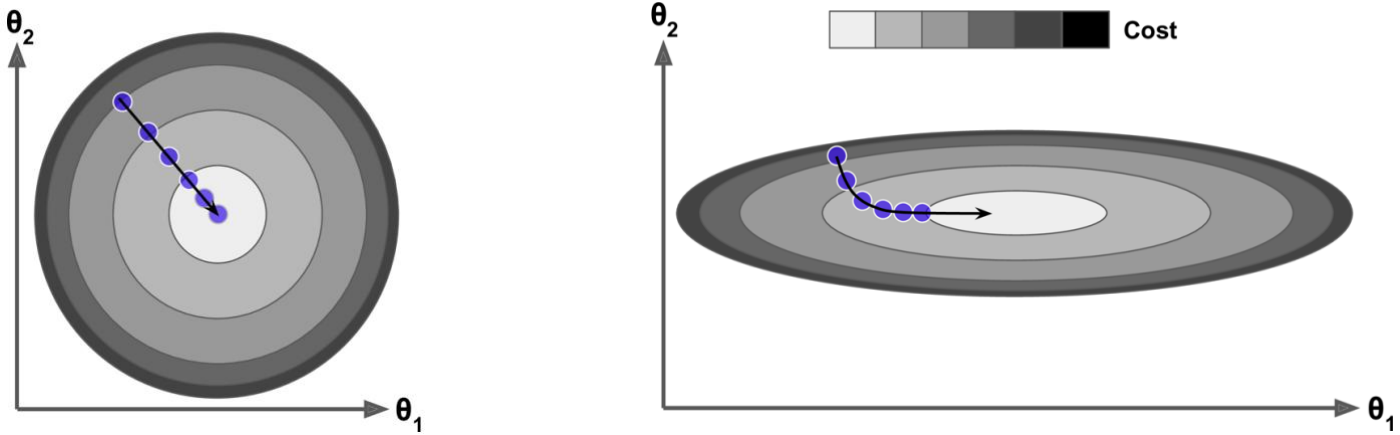
Contour Plot





Effect of Feature Scaling

- When using Gradient Descent, 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.



Gradient Descent with (left) and without (right) feature scaling

Batch Gradient Descent

- To implement Gradient Descent, you need to compute the gradient of the cost function with regard to each model parameter θ_j.
- Take partial derivative of the cost function with regard to θ_j:

$$\mathrm{MSE}\left(\mathbf{X}, h_{\boldsymbol{\theta}}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}^{(i)} - y^{(i)}\right)^2$$

$$\frac{\partial}{\partial\theta_j}\mathrm{MSE}\left(\boldsymbol{\theta}\right) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

$$\nabla_{\boldsymbol{\theta}}\,\mathrm{MSE}\left(\boldsymbol{\theta}\right) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\mathrm{MSE}\left(\boldsymbol{\theta}\right) \\ \frac{\partial}{\partial\theta_1}\mathrm{MSE}\left(\boldsymbol{\theta}\right) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\mathrm{MSE}\left(\boldsymbol{\theta}\right) \end{pmatrix} = \frac{2}{m}\mathbf{X}^{\mathsf{T}}\left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right)$$

Batch Gradient Descent

- Gradient vector involves calculations over the full training set $\mathbf{X}$, at each Gradient Descent step: $\nabla_{\boldsymbol{\theta}} \operatorname{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \mathbf{X}^{\intercal} (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$
  - This is why the algorithm is called *Batch Gradient Descent*.
- Gradient Descent step: $\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \operatorname{MSE}(\boldsymbol{\theta})$

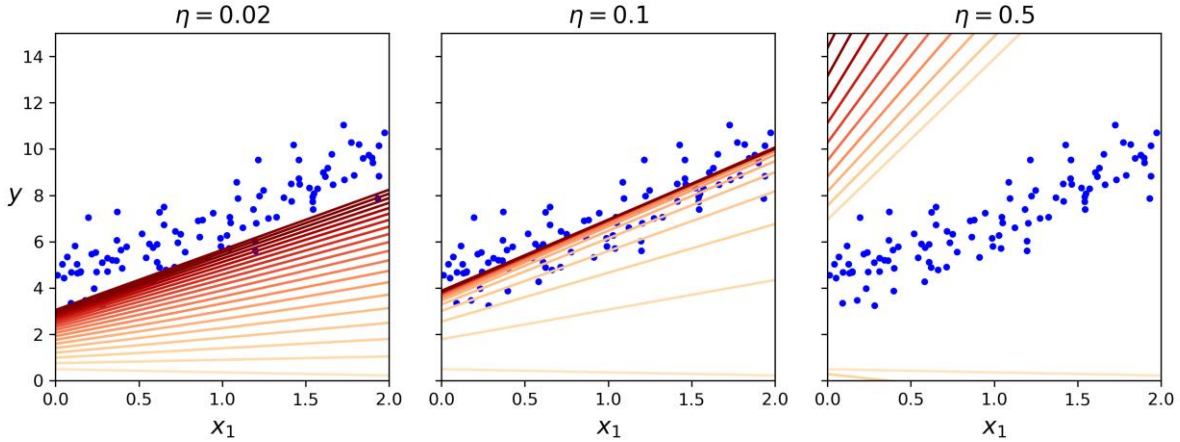
```
eta = 0.1  # learning rate
n_epochs = 1000
m = len(X_b)  # number of instances

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]])
```

Convergence Rate

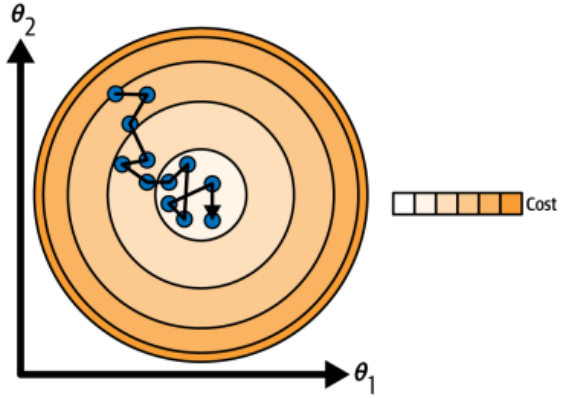
- To find a good learning rate, use grid search with limited iterations.
- Set a very large number of iterations but interrupt the algorithm when the gradient vector norm is smaller than a tolerance ϵ.
- It takes $O(1/\epsilon)$ iterations for Batch Gradient Descent with a fixed learning rate to reach the optimum within a range of ϵ.



Stochastic Gradient Descent

- **Batch** Gradient Descent uses the **whole training set** to compute the gradients at every step, which makes it very slow.
- *Stochastic Gradient Descent (SGD)* picks a random instance in the training set at every step and computes the gradients based only on that single instance.
  - Much faster because it has little data to manipulate at every step.
  - Possible to train on huge training sets, since only one instance needs to be in memory at each iteration (an out-of-core algorithm)
- Stochastic nature of SGD makes it much less regular than Batch Gradient Descent.

Pros and Cons of SGD

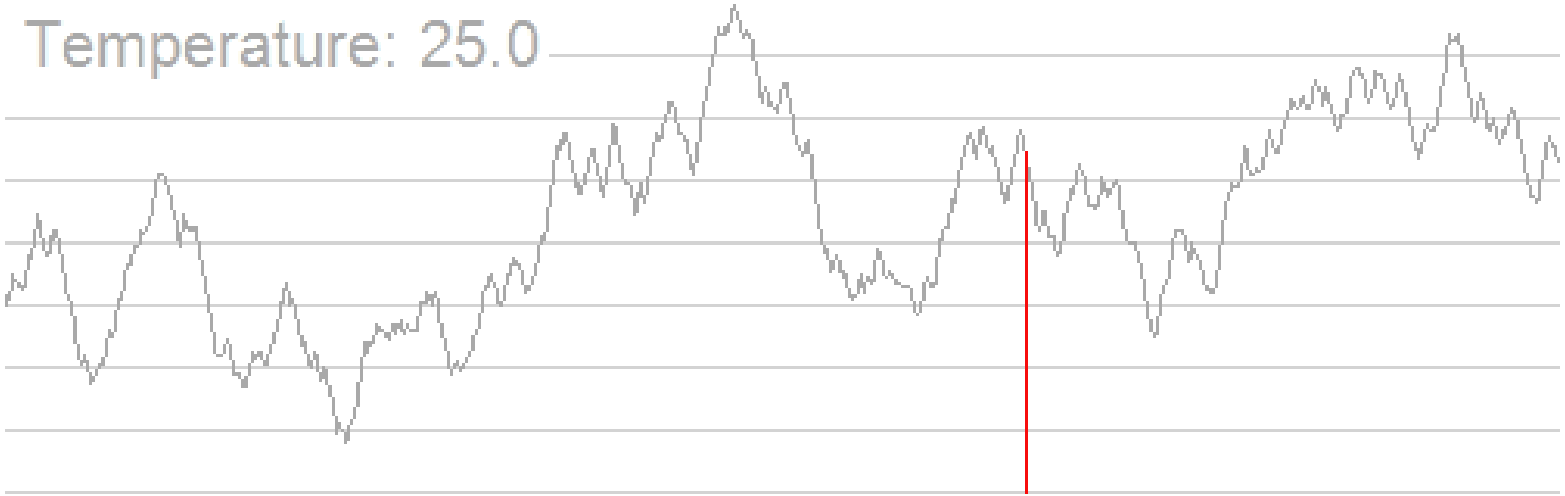


- In SGD the cost function will bounce up and down, *decreasing* only *on average*.
- Once the algorithm stops, the final parameter values are good, but not optimal.
- Irregular changes in the cost function can help the algorithm jump out of local minima, so SGD has a better chance of finding the global minimum.
- The randomness is good to escape from local optima, but bad because the algorithm can never settle at the minimum.

Simulated Annealing

- *Simulated annealing*: gradually reduce the learning rate.
- The steps start out large (helps make quick progress and escape local minima), then get smaller, allowing the algorithm to settle at the global minimum.
- 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.
  - If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution.

Simulated Annealing



Implementation of SGD

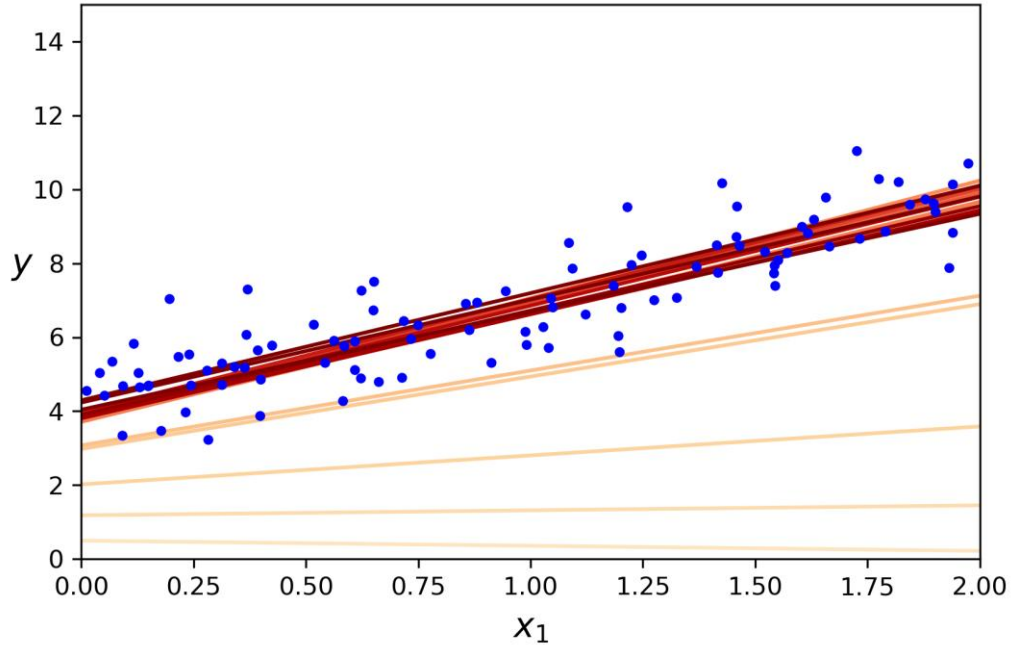
```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

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

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

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

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

Implementation of SGD

- The first 20 steps of Stochastic Gradient Descent:



SGD with Scikit-Learn

- To perform Linear Regression using SGD with Scikit-Learn, you can use the `SGDRegressor` class:

```
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
```

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

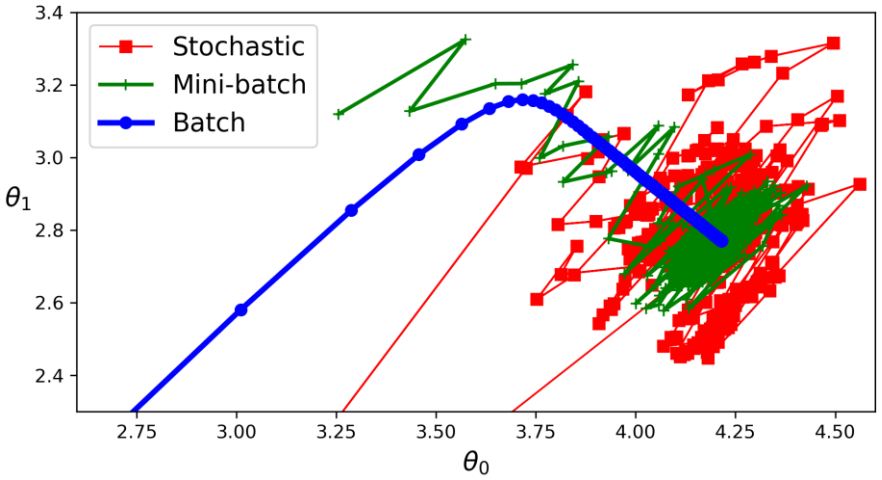
```
(array([4.21278812]), array([2.77270267]))
```

- Scikit-Learn estimators can be trained using the `fit()` method, but some estimators also have a `partial_fit()` method to run a single round of training on one or more instances.

Mini-batch Gradient Descent

- *Mini-batch Gradient Descent*: at each step, compute the gradients on small random sets of instances called *mini-batches*.

- Gets a performance boost from hardware optimization of matrix operations, especially when using GPUs.
- Gets a bit closer to the optimum.
- Harder to escape from local minima.



Comparison of algorithms for linear regression

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal equation	Fast	No	Slow	0	No	N/A
SVD	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