

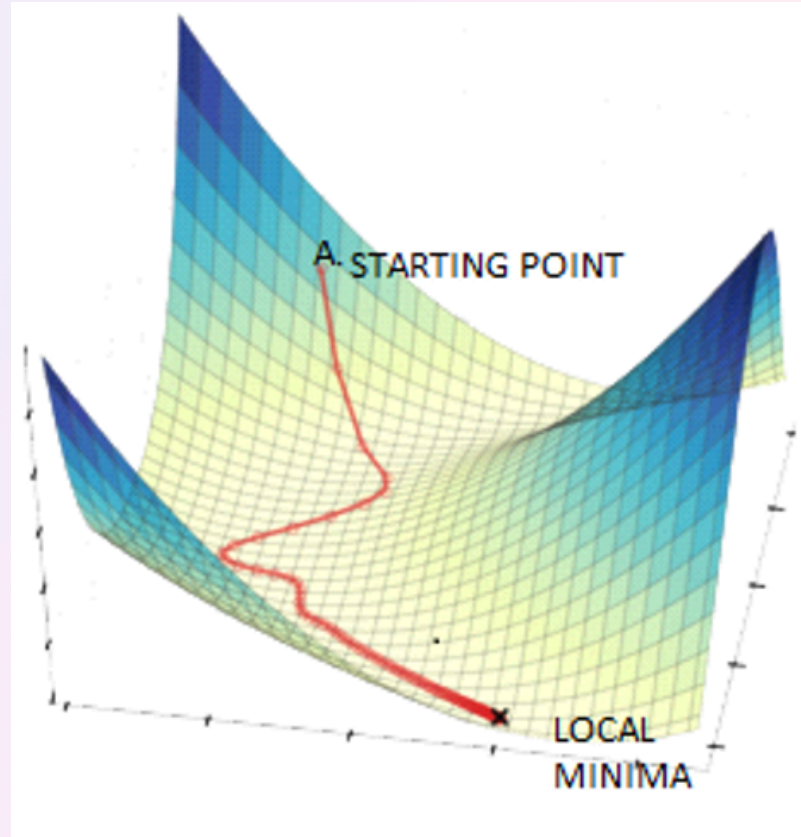
Gradient Descent

David I. Inouye

Most AI/ML Optimizations Must Be **Numerically** Optimized

- One common algorithm is **gradient descent**
 - Primary algorithm for deep learning
 - Works in very high dimensions
- Other optimization algorithms
 - Expectation Maximization (alternating optimization)
 - Sampling-based optimization (MCMC/Gibb)
 - Greedy optimization (e.g., decision trees)

Gradient Descent Is Like Taking Steps Down the Steepest Descent Into a Valley



<https://www.hackerearth.com/blog/developers/3-types-gradient-descent-algorithms-small-large-data-sets/>

Vanilla **Gradient Descent (GD)** Has Simple Form

- Objective (**Loss**) function denoted by $\mathcal{L}(\theta; \mathcal{D})$:

$$\operatorname{argmin}_{\theta} \mathcal{L}(\theta; \mathcal{D})$$

1. Start at random parameter, e.g., $\theta^0 \sim \mathcal{N}(0, 1)$
2. Iteratively update parameter via **negative gradient** of loss function (η_t is step size or learning rate)

$$\theta^{t+1} = \theta^t - \eta_t \nabla_{\theta} \mathcal{L}(\theta^t)$$

- η_t is **learning rate** (or **step size**)

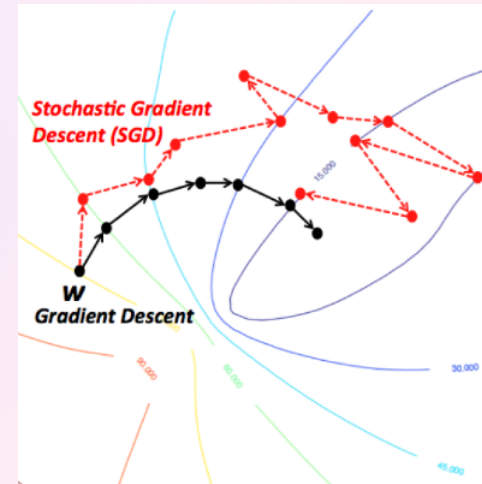
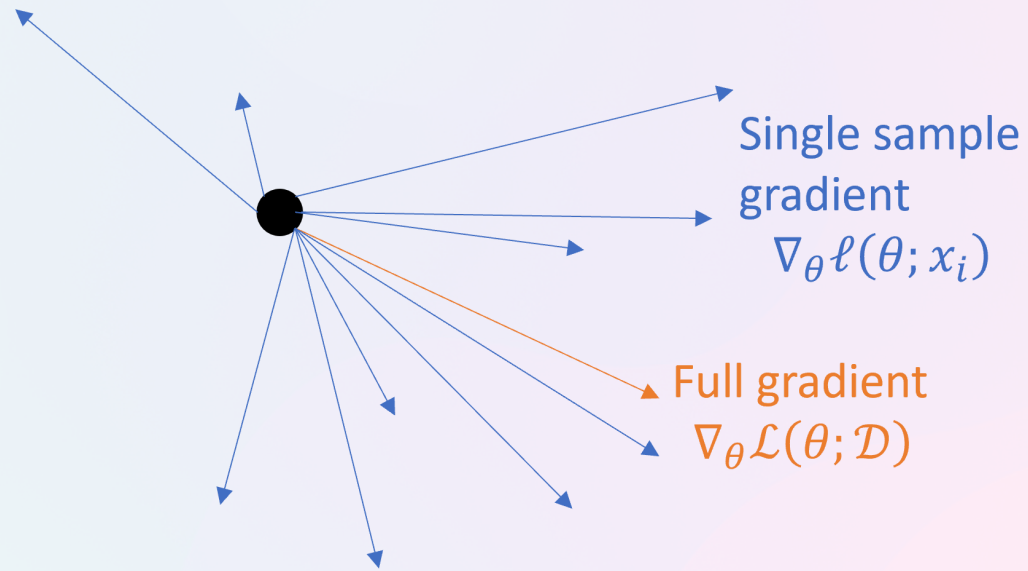
Stochastic Gradient Descent (SGD) Merely Uses One Sample in the Gradient Calculation

- The loss function can usually be split into a summation of losses $l(\theta; x_i)$ for each sample x_i :
 - $\mathcal{L}(\theta; \mathcal{D}) = \frac{1}{n} \sum_{i=1}^n l(\theta; x_i)$
- SGD approximates the full gradient by the gradient of a single sample
 - $\nabla_{\theta} \mathcal{L}(\theta^t; \mathcal{D}) \approx \nabla_{\theta} l(\theta^t; x_i)$
 - Theoretically, $\mathbb{E}_i[\nabla_{\theta} l(\theta^t; x_i)] = \nabla_{\theta} \mathcal{L}(\theta^t; \mathcal{D})$
- Loop through all $x_i \in \mathcal{D}$ (One pass through dataset)

$$\tilde{\theta}^{t+1} = \theta^t - \eta_t \nabla_{\theta} l(\theta^t; x_i)$$

- GD: 1 large update with $O(n)$ cost
- SGD: n smaller updates with $O(1)$ cost each

Stochastic Gradient Descent (SGD) Merely Uses One Sample in the Gradient Calculation



Full gradient is average over single sample gradients. This is why it is “stochastic”.

Mini-Batch SGD (or Just SGD) Uses a Small Batch of Samples in the Gradient Calculation

- **Mini-batch SGD** approximates the full gradient by the gradient of a batch of samples
 - Sample mini-batch

$$\theta^{t+1} = \theta^t - \eta_t \sum_{k=1}^b \frac{1}{b} \nabla_{\theta} l(\theta^t; x_k)$$

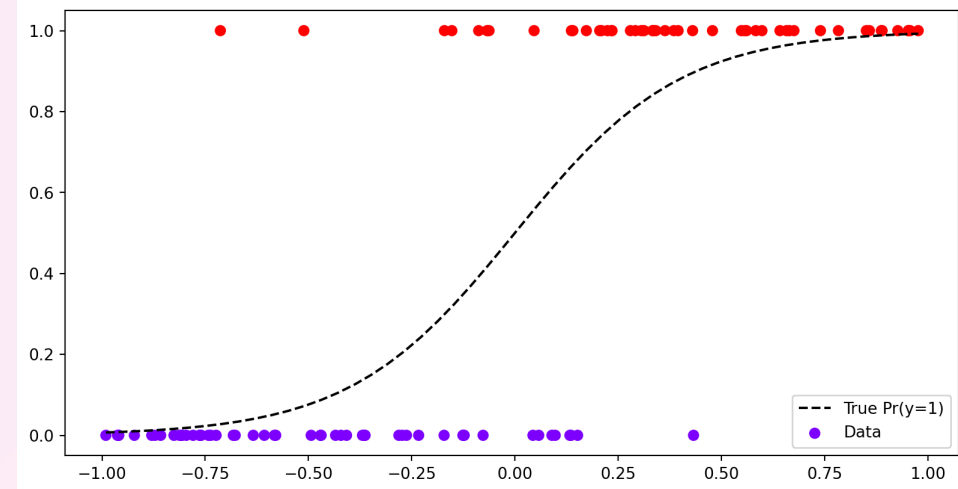
- One pass through dataset
 - GD: 1 large update
 - SGD: n smaller updates
 - Mini-batch SGD: $\frac{n}{b}$ medium-size updates

Gradient Descent Demo for Simplified Logistic Regression

Create Simple Logistic Data and Plot True Model

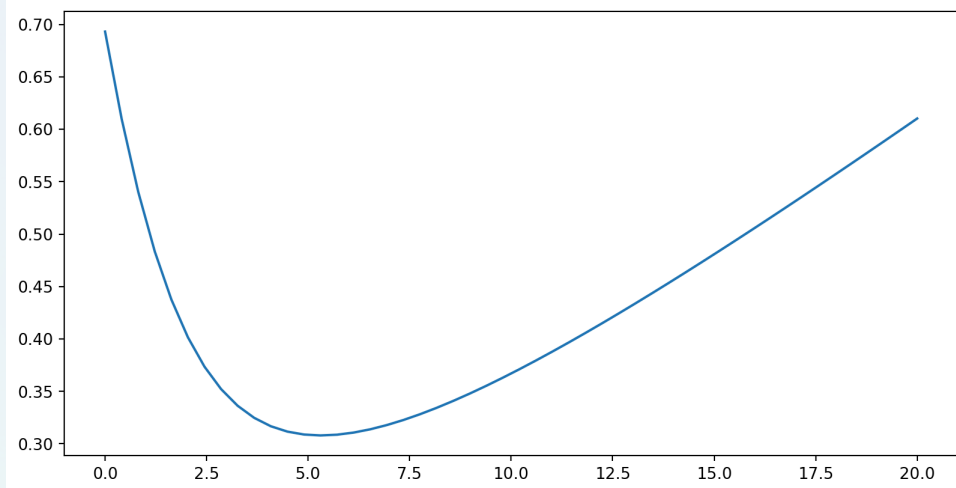
```
1 import numpy as np
2 import scipy.stats
3 import matplotlib.pyplot as plt
4
5 # Create simple logistic data
6 rng = np.random.RandomState(0)
7 n_samples = 100
8 x = 2*(rng.rand(n_samples)-0.5)
9
10 theta_true = 5
11 # Very simplified logistic regression model with only 1 param
12 def model(x, theta):
13     return 1 / (1 + np.exp(-x.dot(theta)))
14 y = (rng.rand(*x.shape) <= model(x, theta_true)).astype(float)
15 print(y.shape)
16 # Plot data
17 xq = np.linspace(np.min(x), np.max(x))
18 plt.plot(xq, model(xq, theta_true), '--k', label='True Pr(y=1)')
19 plt.scatter(x, y, c=y, cmap='rainbow', label='Data')
20 plt.legend()
```

(100,)



Define the Logistic Loss (Negative Log-Likelihood)

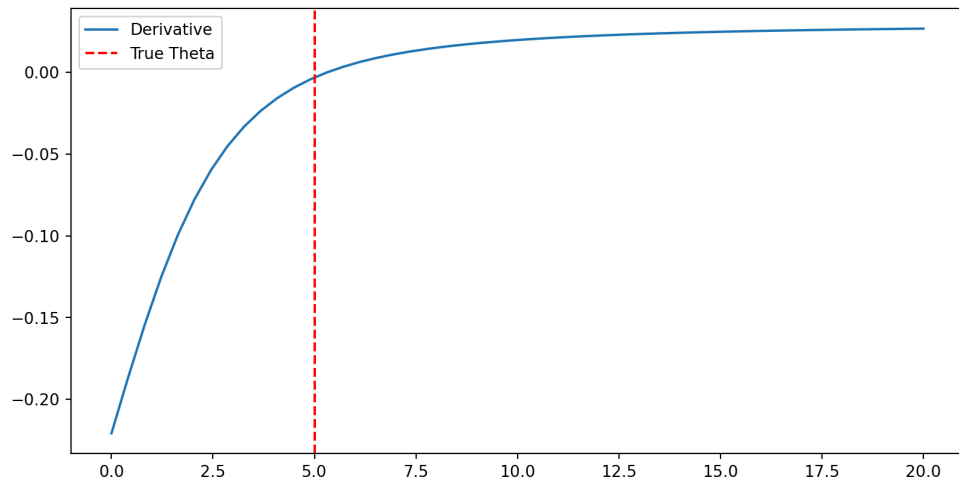
```
1 def objective(x, y, theta):  
2     prob = model(x, theta)  
3     return -np.mean(y * np.log(prob) + (1-y) * np.log(1 - prob))  
4  
5 theta_arr = np.linspace(0, 20)  
6 obj_arr = [objective(x, y, theta) for theta in theta_arr]  
7 plt.plot(theta_arr, obj_arr)
```



NOTE: Because of randomness, empirical estimate will not exactly match true model. (e.g., this has a minimum closer to 5.3 or so)

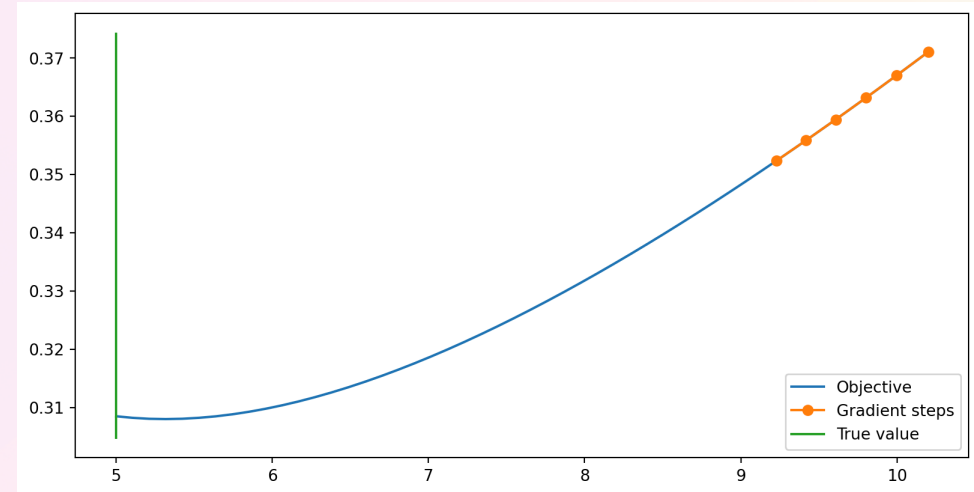
Explicitly compute (by hand) the gradient of the function

```
1 def grad_objective(x, y, theta):
2     #See https://web.stanford.edu/~jurafsky/slp3/5.pdf
3     return np.mean((model(x, theta) - y) * x)
4
5 grad_arr = [grad_objective(x, y, theta) for theta in theta_arr]
6 plt.plot(theta_arr, grad_arr, label='Derivative')
7 plt.axvline(5, linestyle='--', color='r', label='True Theta')
8 plt.legend()
```



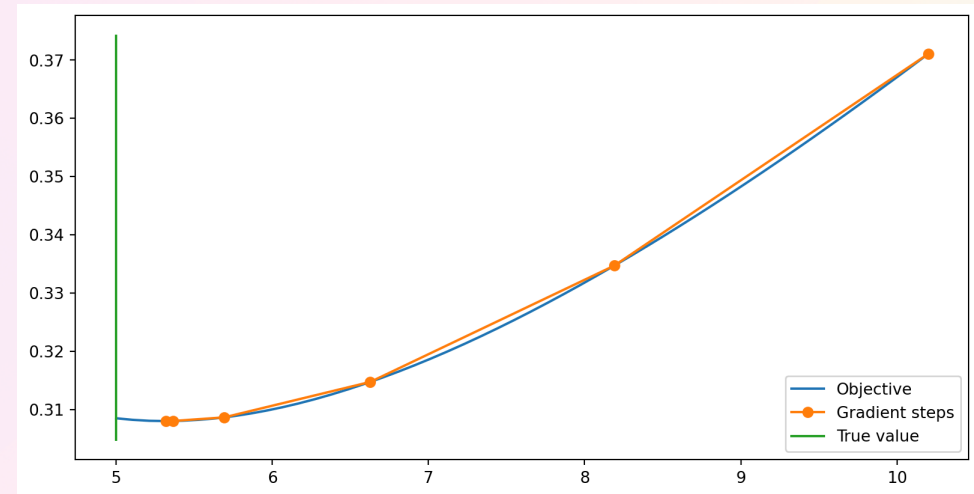
Gradient Descent with various step sizes

```
1 # Gradient descent parameters
2 max_iter = 5
3 step_size = 10
4 sgd = False
5 if sgd: max_iter *= 10 # Increase number of iterations for SGD
6 rng = np.random.RandomState(0)
7
8 # Initialization
9 theta_hat = 10.2 # An arbitrary starting value
10 theta_hat_arr = [theta_hat]
11 obj_hat_arr = [objective(x, y, theta_hat)]
12
13 # Gradient descent iterations
14 for it in range(max_iter):
15     if sgd:
16         # Select random data point
17         rand_idx = rng.randint(len(y))
18         xg, yg = x[rand_idx:rand_idx+1], y[rand_idx:rand_idx+1]
19     else:
20         # Use all data points in gradient calculation
21         xg, yg = x, y
22     grad = grad_objective(xg, yg, theta_hat)
23     theta_hat = theta_hat - step_size * grad
24
25     # Save estimates for visualization
26     theta_hat_arr.append(theta_hat)
27     obj_hat_arr.append(objective(x, y, theta_hat))
```



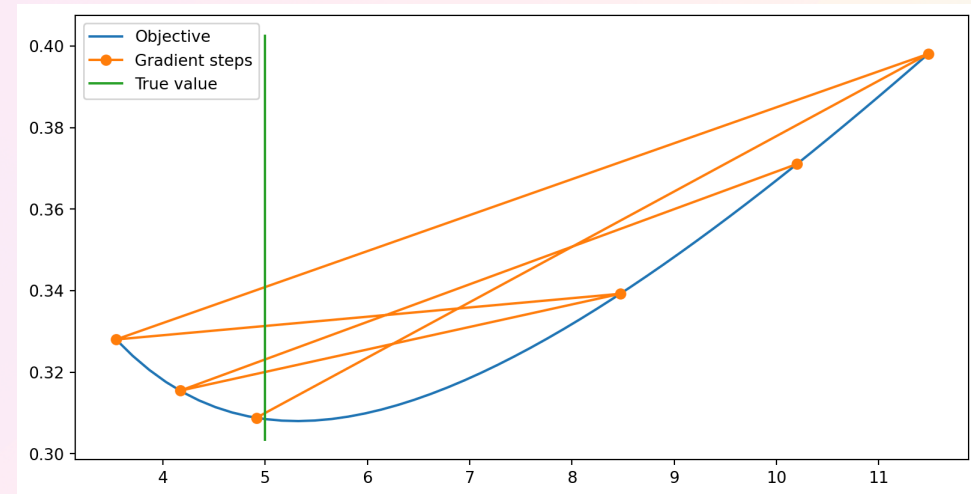
Gradient Descent with various step sizes

```
1 # Gradient descent parameters
2 max_iter = 5
3 step_size = 100
4 sgd = False
5 if sgd: max_iter *= 10 # Increase number of iterations for SGD
6 rng = np.random.RandomState(0)
7
8 # Initialization
9 theta_hat = 10.2 # An arbitrary starting value
10 theta_hat_arr = [theta_hat]
11 obj_hat_arr = [objective(x, y, theta_hat)]
12
13 # Gradient descent iterations
14 for it in range(max_iter):
15     if sgd:
16         # Select random data point
17         rand_idx = rng.randint(len(y))
18         xg, yg = x[rand_idx:rand_idx+1], y[rand_idx:rand_idx+1]
19     else:
20         # Use all data points in gradient calculation
21         xg, yg = x, y
22     grad = grad_objective(xg, yg, theta_hat)
23     theta_hat = theta_hat - step_size * grad
24
25     # Save estimates for visualization
26     theta_hat_arr.append(theta_hat)
27     obj_hat_arr.append(objective(x, y, theta_hat))
```



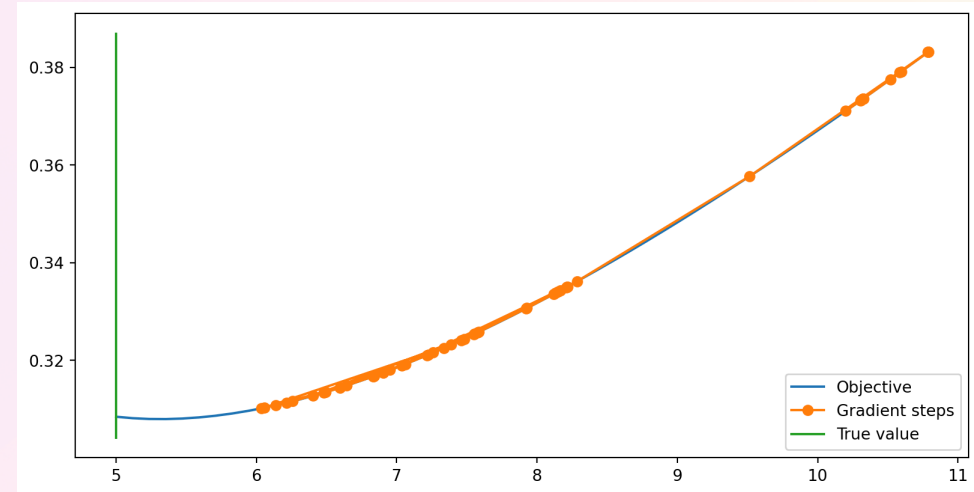
Gradient Descent with various step sizes

```
1 # Gradient descent parameters
2 max_iter = 5
3 step_size = 300
4 sgd = False
5 if sgd: max_iter *= 10 # Increase number of iterations for SGD
6 rng = np.random.RandomState(0)
7
8 # Initialization
9 theta_hat = 10.2 # An arbitrary starting value
10 theta_hat_arr = [theta_hat]
11 obj_hat_arr = [objective(x, y, theta_hat)]
12
13 # Gradient descent iterations
14 for it in range(max_iter):
15     if sgd:
16         # Select random data point
17         rand_idx = rng.randint(len(y))
18         xg, yg = x[rand_idx:rand_idx+1], y[rand_idx:rand_idx+1]
19     else:
20         # Use all data points in gradient calculation
21         xg, yg = x, y
22     grad = grad_objective(xg, yg, theta_hat)
23     theta_hat = theta_hat - step_size * grad
24
25     # Save estimates for visualization
26     theta_hat_arr.append(theta_hat)
27     obj_hat_arr.append(objective(x, y, theta_hat))
```



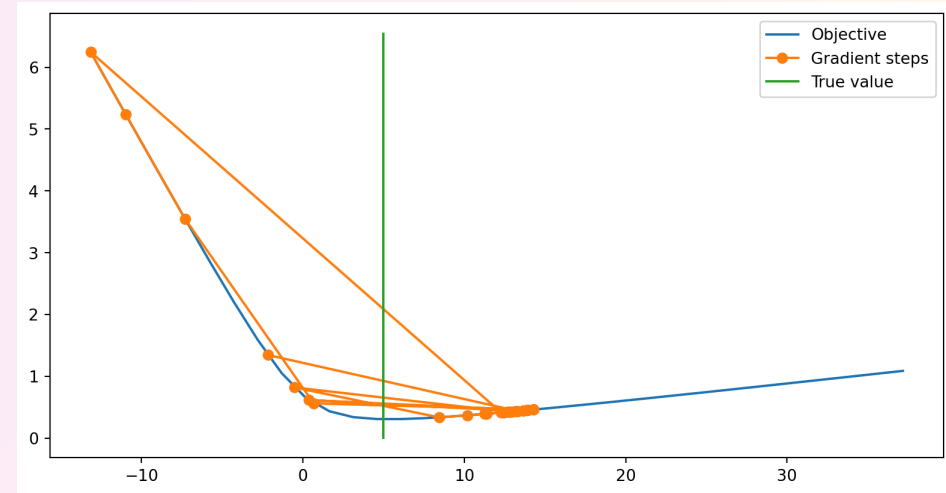
SGD with various step sizes

```
1 # Gradient descent parameters
2 max_iter = 5
3 step_size = 10
4 sgd = True
5 if sgd: max_iter *= 10 # Increase number of iterations for SGD
6 rng = np.random.RandomState(0)
7
8 # Initialization
9 theta_hat = 10.2 # An arbitrary starting value
10 theta_hat_arr = [theta_hat]
11 obj_hat_arr = [objective(x, y, theta_hat)]
12
13 # Gradient descent iterations
14 for it in range(max_iter):
15     if sgd:
16         # Select random data point
17         rand_idx = rng.randint(len(y))
18         xg, yg = x[rand_idx:rand_idx+1], y[rand_idx:rand_idx+1]
19     else:
20         # Use all data points in gradient calculation
21         xg, yg = x, y
22     grad = grad_objective(xg, yg, theta_hat)
23     theta_hat = theta_hat - step_size * grad
24
25     # Save estimates for visualization
26     theta_hat_arr.append(theta_hat)
27     obj_hat_arr.append(objective(x, y, theta_hat))
```



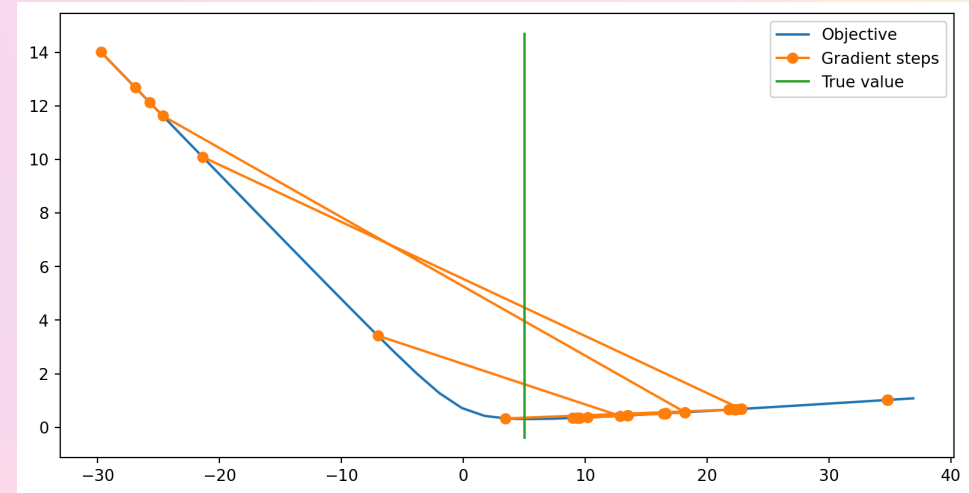
SGD with various step sizes

```
1 # Gradient descent parameters
2 max_iter = 5
3 step_size = 100
4 sgd = True
5 if sgd: max_iter *= 10 # Increase number of iterations for SGD
6 rng = np.random.RandomState(0)
7
8 # Initialization
9 theta_hat = 10.2 # An arbitrary starting value
10 theta_hat_arr = [theta_hat]
11 obj_hat_arr = [objective(x, y, theta_hat)]
12
13 # Gradient descent iterations
14 for it in range(max_iter):
15     if sgd:
16         # Select random data point
17         rand_idx = rng.randint(len(y))
18         xg, yg = x[rand_idx:rand_idx+1], y[rand_idx:rand_idx+1]
19     else:
20         # Use all data points in gradient calculation
21         xg, yg = x, y
22     grad = grad_objective(xg, yg, theta_hat)
23     theta_hat = theta_hat - step_size * grad
24
25     # Save estimates for visualization
26     theta_hat_arr.append(theta_hat)
27     obj_hat_arr.append(objective(x, y, theta_hat))
```



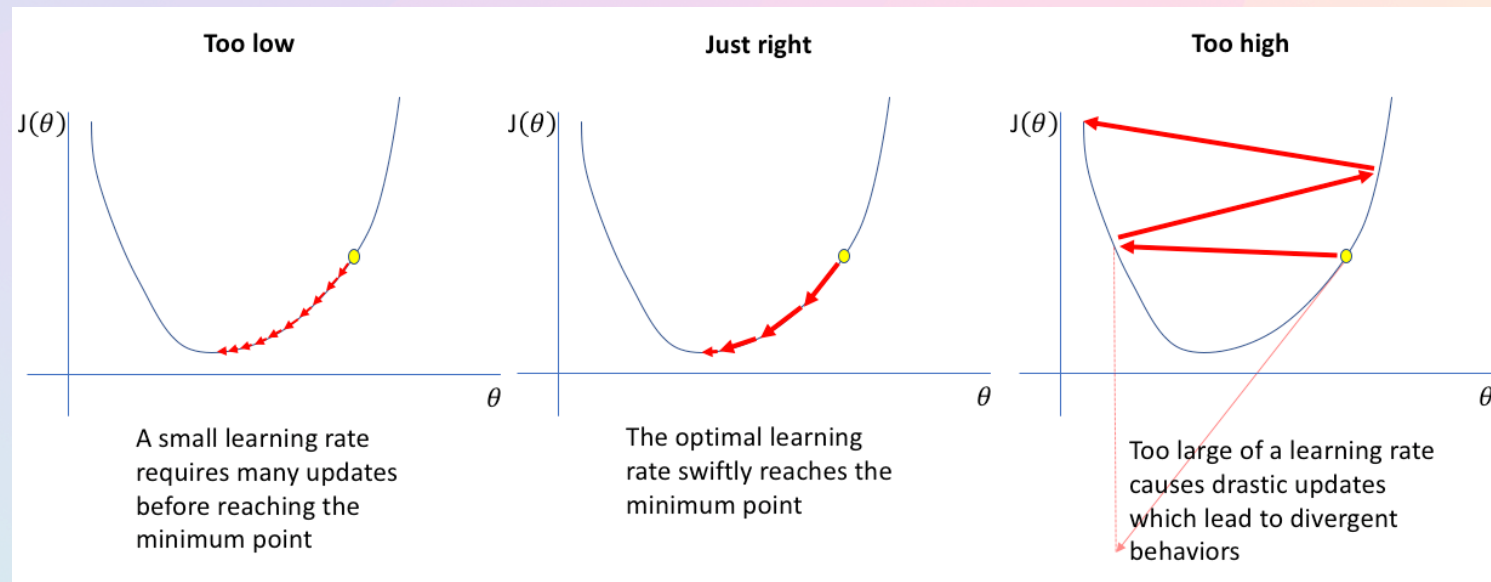
SGD with various step sizes

```
1 # Gradient descent parameters
2 max_iter = 5
3 step_size = 300
4 sgd = True
5 if sgd: max_iter *= 10 # Increase number of iterations for SGD
6 rng = np.random.RandomState(0)
7
8 # Initialization
9 theta_hat = 10.2 # An arbitrary starting value
10 theta_hat_arr = [theta_hat]
11 obj_hat_arr = [objective(x, y, theta_hat)]
12
13 # Gradient descent iterations
14 for it in range(max_iter):
15     if sgd:
16         # Select random data point
17         rand_idx = rng.randint(len(y))
18         xg, yg = x[rand_idx:rand_idx+1], y[rand_idx:rand_idx+1]
19     else:
20         # Use all data points in gradient calculation
21         xg, yg = x, y
22     grad = grad_objective(xg, yg, theta_hat)
23     theta_hat = theta_hat - step_size * grad
24
25     # Save estimates for visualization
26     theta_hat_arr.append(theta_hat)
27     obj_hat_arr.append(objective(x, y, theta_hat))
```



Learning Rate / Step Size Is Critical for Convergence and Correctness of Algorithm

- If learning rate is **too high**, the algorithm could **diverge**.
 - Diverge means to get farther away from the solution.
- If learning rate **too low**, the algorithm could take a very long time to converge.



Adaptive Learning Rates May Help (**but Not Always**)

- Decreasing step size, $\eta_t = \frac{1}{t}$
 - Intuition: Approaches 0 but can cover an infinite distance since $\lim_{a \rightarrow \infty} \sum_{t=1}^a \frac{1}{t} = \infty$
- ADAM - Adaptive Moment Estimation
- See <https://pytorch.org/docs/stable/optim.html> for more options

Parameter Initialization Can Be Important if Non-Convex or Step Size Incorrect

- If convex function, initial parameter θ^0 **will not** affect final optimization result $\theta = \operatorname{argmin} \mathcal{L}(\theta)$.
 - Yay! (Assuming appropriate step size.)
- If non-convex, starting position **WILL** affect final converged θ .
 - Sad day. (But sometimes it's not too bad in practice.)

