# Chapter 2 Extensions

Policy Gradient Theorem

## Gradient Ascent With an Unbiased Estimator

Let's illustrate how an unbiased estimator can replace the true gradient and still push the parameters toward the maximum. We will optimize the same objective function $J(\theta_1, \theta_2) = 10 - (\theta_1 - 2)^2 - (\theta_2 - 3)^2$ from earlier.

For this objective, the exact gradient is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \begin{bmatrix} -2(\theta_1 - 2) \\ -2(\theta_2 - 3) \end{bmatrix}.$$

To isolate the effect of stochastic estimation, we use this analytic gradient and add random noise to it. The noise follows a Gaussian with mean zero:

$$\widehat{\boldsymbol{G}} \overset{\text{def}}{=} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \boldsymbol{\varepsilon}, \qquad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2 I).$$

The notation $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2 I)$ shows where the random noise comes from: the tilde "$\sim$" reads as "is distributed as," and $\mathcal{N}$ denotes the Gaussian distribution, with mean $\mathbf{0}$ (so the noise averages to zero) and variance controlled by $\sigma^2$. We take the true gradient and add zero-mean random noise to it.

This estimate is unbiased because its expected value equals the exact gradient. The expectation of a sum is the sum of expectations, so

$$\mathbb{E}[\widehat{\boldsymbol{G}}] = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \mathbb{E}[\boldsymbol{\varepsilon}] = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \mathbf{0} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}).$$

The noise averages out because its mean is zero. Any single $\widehat{\boldsymbol{G}}$ is off in some direction, but on average it points exactly where the true gradient points. As we established, this is what lets gradient ascent keep making progress toward the maximum, even though each individual step is noisy.

> Code by hand or use a chatbot to create a Python script that generates vectors from $\mathcal{N}(\mathbf{0}, \sigma^2 I)$ and averages them to confirm that the average converges to [0,0].

Now, let's update the code. We first implement the analytic gradient of the objective, then add zero-mean Gaussian noise to it:

```
def compute_gradient(theta):
    theta1, theta2 = theta
    return np.array([
        -2 * (theta1 - 2),
        -2 * (theta2 - 3),
    ])

def estimate_gradient(theta, sigma=1.0, rng=None):
    if rng is None:
        rng = np.random.default_rng()
    exact = compute_gradient(theta) ❶
    noise = sigma * rng.standard_normal(size=theta.shape) ❷
    return exact + noise ❸
```

In the estimate_gradient:

- In line ❶, we compute the analytic gradient defined above.
- In line ❷, we generate zero-mean Gaussian noise, one number per parameter, scaled by sigma.

- In line ❸, we return the exact gradient plus the noise. This sum is our unbiased estimate $\widehat{\boldsymbol{G}}$: each call is noisy, but its expected value is the exact gradient.

The ascent loop is unchanged except that the gradient now comes from estimate_gradient:

```
def gradient_ascent(theta, learning_rate, num_iterations, rng):
    for _ in range(num_iterations):
        grad = estimate_gradient(theta, rng=rng) ❶
        theta += learning_rate * grad ❷
    return theta
```

In the above function:

- In line ❶, we get an unbiased estimate of the gradient.
- In line ❷, we apply the usual update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\widehat{\boldsymbol{G}}$.

Finally, we set up the run. We create a random number generator with a fixed seed, so the noise, and therefore the result, is the same every time the script runs:

```
if __name__ == '__main__':
    rng = np.random.default_rng(0) ❶
    theta_start = np.array([3.8, 1.5])
    learning_rate = 0.1
    max_iterations = 30

    theta_optimal = gradient_ascent( ❷
        theta_start,
        learning_rate,
        max_iterations,
        rng
    )

    optimal_value = objective_function(theta_optimal) ❸
    print(f"Optimal theta: {theta_optimal}")
    print(f"Objective function value at optimal theta: {optimal_value}")
```

In the above code block:

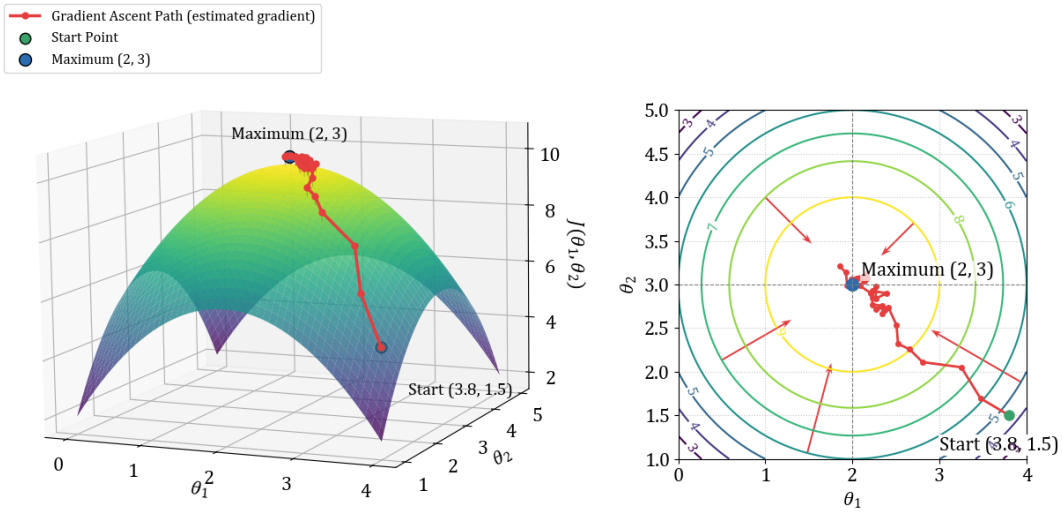
- In line ❶, we create a random number generator with a fixed seed, 0. The seed determines the exact sequence of noise values, so the script produces the same result every time it runs. Changing the seed gives a different noise sequence and a slightly different result.
- In line ❷, we pass the generator to gradient_ascent, which hands it to estimate_gradient on every iteration. Using one generator for the whole run, instead of creating a new one on each call, is what makes the run reproducible.
- In line ❸, we evaluate the objective at the final parameters using the objective_function we defined for the exact-gradient version in Chapter 1. The starting point, learning rate and number of iterations are also the same as before, so the only difference between the two runs is the noise in the gradient.

If you run this script, you should see:

```
Optimal theta: [1.95208214 3.1389554]
Objective function value at optimal theta: 9.978395274681816
```

The exact numbers depend on the random noise, so they differ from the near-perfect [2.00222829 2.99814309] of the exact-gradient version, and they change if you change the seed or sigma. What matters is the overall movement: although every step used a corrupted gradient, the parameters still traveled from [3.8,1.5] into a small neighborhood of the true maximum at [2,3], where $J$ is close to 10.

You can see the process of moving towards the maximum with an estimate of the gradient on the plot below:




This is the behavior we relied on when we argued that an unbiased estimate of the gradient can stand in for the real thing. Because the noise averages to zero, it does not pull the path away from the maximum on average. It does, however, keep the path from settling exactly on the maximum. With a fixed learning rate, every step adds fresh noise, so once the parameters reach the neighborhood of [2,3], they keep jittering around it instead of converging to it. A smaller learning rate or a less noisy estimate shrinks this jitter. Gradually decreasing the learning rate over the course of training lets the parameters converge to the maximum itself.