Chapter 1. Introduction to Reinforcement Learning

Gradients and Gradient Ascent

Gradient Ascent in Python

Let's implement the gradient ascent algorithm to find the maximum of the example objective function $J([\theta_1, \theta_2]) = 10 - (\theta_1 - 2)^2 - (\theta_2 - 3)^2$ from earlier.

First, we import the numpy library, which is commonly used for numerical operations in Python, especially for working with arrays, which we'll use for our parameter vector $\boldsymbol{\theta}$.

```
import numpy as np
```

The next step is to define the function we want to maximize. In our example, this is $J(\theta_1, \theta_2) = 10 - (\theta_1 - 2)^2 - (\theta_2 - 3)^2$. This function takes a parameter vector theta and returns the value of J at that point.

```
def objective_function(theta):
    theta1, theta2 = theta ❶
    return 10 - (theta1 - 2)**2 - (theta2 - 3)**2 ❷
```

In line ❶, the input theta is expected to be a NumPy array (or a list) containing two values; these are unpacked into theta1 and theta2. In line ❷, the function computes and returns the value of $J(\theta_1, \theta_2)$ using the unpacked parameters.

Now, we need a way to compute the gradient $\nabla J(\boldsymbol{\theta})$. We'll use the **finite difference** method, specifically the **central difference** formula, to approximate the partial derivatives:

$$\frac{\partial J}{\partial \theta_i} \approx \frac{J([\theta_1, \ldots, \theta_i + h, \ldots, \theta_d]) - J([\theta_1, \ldots, \theta_i - h, \ldots, \theta_d])}{2h},$$

where h is a very small number.

The compute_gradient function implements this:

```
def compute_gradient(obj, theta, h=1e-5):
    grad = np.zeros_like(theta, dtype=float) ❶

    for i in range(len(theta)):
        theta_plus_h = theta.copy() ❷
        theta_plus_h[i] += h

        theta_minus_h = theta.copy() ❷
        theta_minus_h[i] -= h

        grad[i] = (obj(theta_plus_h) - obj(theta_minus_h)) / (2 * h) ❸

    return grad
```

In the above function:

- In line ❶, a NumPy array grad is initialized with the same shape as theta and filled with zeros. This will store the computed partial derivatives. dtype=float ensures floating-point precision.
- In line ❷, for each parameter θ_i, independent copies (theta_plus_h and theta_minus_h) of the current theta vector are made. This is crucial so that perturbing one component for the finite difference calculation does not affect the original theta or subsequent calculations for other components. The variable theta_plus_h[i] is incremented by h, while theta_minus_h[i] is decremented by h.

- In line ❸, the i-th component of the gradient, $\frac{\partial J}{\partial \theta_i}$, is approximated using the central difference formula. It uses the objective function obj evaluated at the perturbed parameter vectors.

Now, we implement the main gradient ascent loop. This function will take an initial guess for $\boldsymbol{\theta}$, a learning rate α, the number of iterations to run, and the objective function:

```
def gradient_ascent(theta, learning_rate, num_iterations, obj):
    for _ in range(num_iterations): ❶
        grad = compute_gradient(obj, theta) ❷
        theta += learning_rate * grad ❸

    return theta
```

In the above function:

- In line ❶, gradient ascent iterates num_iterations times, performing an update to theta in each iteration.
- In each iteration:
    - In line ❷, the gradient of the objective function obj is computed at the current parameter values theta.
    - In line ❸, the theta vector is updated according to the gradient ascent rule.

Finally, we set up the parameters for our example and run the gradient_ascent function.

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

    theta_optimal = gradient_ascent(
        theta_start,
        learning_rate,
        max_iterations,
        objective_function
    )

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

In the above code block, the theta_start variable defines the initial guess for the parameters $\boldsymbol{\theta}_0$; learning_rate determines the step size α taken in the direction of the gradient; max_iterations specifies how many update steps the algorithm will perform.

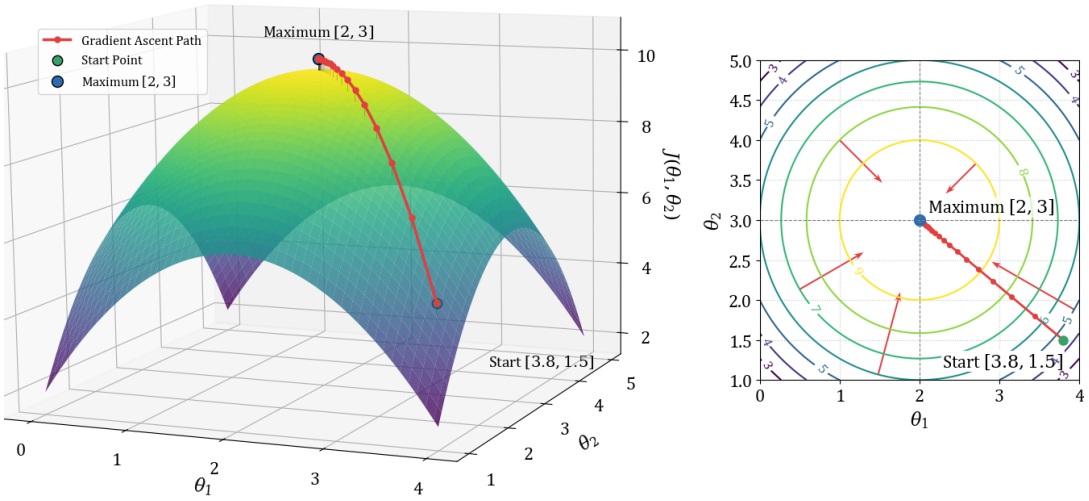
The gradient_ascent function is called with the initial parameters, learning rate, number of iterations, and the objective_function we defined earlier. The result, that is the parameters that maximize the function, is stored in theta_optimal and then printed to the console.

If you run this script, you should see an output similar to:

```
Optimal theta: [2.00222829 2.99814309]
Objective function value at optimal theta: 9.999991586599467
```

This demonstrates that the algorithm has successfully found parameters very close to the true maximum at [2,3], where the function value is 10.

If we apply our gradient ascent code to the function in Figure 1.2, we will see the following gradient ascent path:



The left panel shows a 3D surface representation of the objective function $J(\theta) = 10 - (\theta_1 - 2)^2 - (\theta_2 - 3)^2$, which has a global maximum at $[\theta_1, \theta_2] = [2,3]$ with value $J = 10$. The gradient ascent path (red) starts from the initial point [3.8,1.5] and follows the direction of steepest ascent indicated by the gradient of J. The right panel shows the same optimization in a contour plot view, where concentric circles represent level curves of constant J values. Red arrows indicate the gradient direction at various points, perpendicular to the contour lines and pointing toward increasing values of the objective function.

The Python code above is designed for clarity and illustration. In practice, especially for complex problems like training large neural networks, several aspects would be handled differently for efficiency and robustness. This is not the code we would typically employ for complex, practical reinforcement learning tasks.

Our code's reliance on numerical gradient computation is computationally expensive and can be imprecise for the high-dimensional parameter spaces common in deep reinforcement learning (that is, when we train policies with many layers).

Additionally, using a fixed learning rate is often suboptimal compared to more advanced techniques. When we tackle actual reinforcement learning tasks later, we will leverage more powerful and efficient tools, such as PyTorch, which provides automatic differentiation for efficient gradient calculations. We'll also explore more sophisticated optimization algorithms that often include adaptive learning rate schemes for effective training.

These considerations highlight the difference between a basic illustrative implementation and a production-ready optimization routine used in cutting-edge machine learning. However, the core principle of iteratively moving in the direction of the gradient remains fundamental.

Gradient Ascent in PyTorch

The implementation we just built relies on the finite difference method to *approximate* the gradient. As we noted, this is both computationally expensive and imprecise in high dimensions. We will now rebuild the same gradient ascent procedure using **PyTorch**, a numerical computing library that can calculate gradients automatically, without us ever writing out a difference formula. Before we can do that, we need to introduce two ideas that PyTorch is built around: the **tensor** and **automatic differentiation**.

A **tensor** is PyTorch's basic container for numbers. For our purposes, a tensor is simply PyTorch's counterpart to the NumPy array we used earlier: a one-dimensional tensor holds a vector of numbers, a two-dimensional tensor holds a matrix, and so on. (The name comes from the mathematical generalization of vectors and matrices to any number of dimensions, but we will only need vectors here.) Wherever the NumPy version stored our parameter

vector $\boldsymbol{\theta}$ in an array, the PyTorch version will store it in a tensor instead. We create one by passing a list of numbers to torch.tensor:

```
import torch

theta = torch.tensor([3.8, 1.5])
```

The reason we bother with tensors rather than plain arrays is the second idea: **automatic differentiation** (often shortened to *autograd*). The principle is that PyTorch can watch every arithmetic operation we perform on a tensor and build up a record of how each result was computed from its inputs. This record is called a **computation graph**: a step-by-step history linking the final output back to the original input through every addition, subtraction, and multiplication along the way. Once that history exists, PyTorch can run it in reverse and apply the chain rule of differentiation to work out the derivative of the output with respect to the input. In other words, instead of changing each parameter by a tiny amount h and measuring how the output changes (as the finite difference method does), PyTorch derives the gradient analytically from the operations themselves.

We do not get this tracking by default; we have to ask for it. When we create the tensor holding our parameters, we set the flag requires_grad=True, which tells PyTorch to start recording operations involving that tensor:

```
theta = torch.tensor([3.8, 1.5], requires_grad=True)
```

With those two concepts in hand, let's reimplement gradient ascent. We begin with the objective function $J(\theta_1, \theta_2) = 10 - (\theta_1 - 2)^2 - (\theta_2 - 3)^2$. The body is almost identical to the NumPy version; the only change is that theta is now a tensor, and the arithmetic we perform on it is what PyTorch records in the computation graph.

```
def objective_function(theta):
    theta1, theta2 = theta[0], theta[1]  ❶
    return 10 - (theta1 - 2)**2 - (theta2 - 3)**2  ❷
```

In line ❶, we read the two components out of the theta tensor by indexing into it (theta[0] and theta[1]). Unpacking with theta1, theta2 = theta, as we did in the NumPy version, would work just as well. PyTorch records indexing and unpacking alike, so theta1 and theta2 stay connected to theta in the computation graph either way. In line ❷, the value we return is itself a tensor that carries the record of how it was built from theta, which is what lets PyTorch differentiate it afterward.

Now, instead of writing a compute_gradient function based on finite differences, we let PyTorch's automatic differentiation do the work. The mechanism has two parts. First, evaluating the objective on a tensor that has requires_grad=True produces a result tensor carrying the full computation graph. Second, calling the method .backward() on that result tells PyTorch to run the graph in reverse; as it does so, it computes the derivative of the result with respect to each tracked parameter and stores those derivatives in a .grad attribute attached to the parameter tensor. So once .backward() has been called on the objective's output, the gradient we want is sitting in theta.grad.

We wrap this in a small helper:

```
def compute_gradient(obj, theta):
    if theta.grad is not None:  ❶
        theta.grad.zero_()

    value = obj(theta)  ❷
    value.backward()  ❸

    return theta.grad  ❹
```

In the above function:

- In line ❶, we reset any previously accumulated gradient. PyTorch accumulates gradients into .grad by default (it adds to whatever is already there rather than overwriting it), so before each new computation we must zero it out. On the very first call theta.grad is still None, so we guard against that.

- In line ❷, we evaluate the objective function at the current theta. Because theta was created with gradient tracking enabled, the resulting value carries the computation graph needed for differentiation.
- In line ❸, the call to value.backward() triggers automatic differentiation. PyTorch runs the recorded graph in reverse, starting from value, and computes $\frac{\partial J}{\partial \theta_i}$ for each component, storing the results in theta.grad. Note that no finite difference, no step size h, and no manual formula appear anywhere; the derivatives are obtained analytically through the chain rule.
- In line ❹, we return the gradient tensor that backward() has just filled in.

The main gradient ascent loop mirrors the NumPy version, with one important addition related to how PyTorch tracks operations:

```
def gradient_ascent(theta, learning_rate, num_iterations, obj):
    for _ in range(num_iterations):  ❶
        grad = compute_gradient(obj, theta)  ❷

        with torch.no_grad():  ❸
            theta += learning_rate * grad  ❹

    return theta
```

In the above function:

- In line ❶, just as before, we iterate num_iterations times, updating theta on each pass.
- In line ❷, we obtain the exact gradient at the current parameters using the helper defined above.
- In line ❸, we wrap the parameter update in a torch.no_grad() context. This temporarily switches off the operation-recording we described earlier. The update to theta is part of our optimization procedure, not part of the function whose gradient we want, so we deliberately keep it out of the computation graph. Without this, PyTorch would try to record the update too, and because theta is a tracked parameter that we are modifying in place, it would stop with a RuntimeError on the very first iteration.
- In line ❹, we apply the gradient ascent update rule, taking a step of size learning_rate in the direction of the gradient, as in the NumPy implementation.

Finally, we set up the parameters and run the procedure. The one structural difference from the NumPy setup is that the initial theta must be created as a tensor with requires_grad=True, the flag we introduced earlier that switches on gradient tracking from the very start.

```
if __name__ == '__main__':
    theta_start = torch.tensor([3.8, 1.5], requires_grad=True)  ❶
    learning_rate = 0.1
    max_iterations = 30

    theta_optimal = gradient_ascent(
        theta_start,
        learning_rate,
        max_iterations,
        objective_function
    )

    with torch.no_grad():  ❷
        optimal_value = objective_function(theta_optimal)

    print(f"Optimal theta: {theta_optimal.detach().numpy()}")  ❸
    print("Objective function value at optimal theta: "
          f"{optimal_value.item()}")
```

In the above code block:

- In line ❶, theta_start is created as a tensor with requires_grad=True. The learning_rate and max_iterations play the same roles (α and the number of update steps K) as in the NumPy version.
- In line ❷, we evaluate the final objective value inside a torch.no_grad() block, since at this point we only want the number itself and have no reason to record operations.
- In line ❸, we convert the resulting tensors back into ordinary numbers for printing. The .detach() call separates theta_optimal from its computation graph before we turn it into a NumPy array, and .item() pulls the single scalar number out of the objective tensor.

If you run this script, you should see an output similar to:

```
Optimal theta: [2.0022283 2.9981432]
Objective function value at optimal theta: 9.999991416931152
```

The result matches the NumPy implementation to roughly six significant figures, converging to the true maximum at $[2,3]$ where $J = 10$. The two outputs are not identical: by default PyTorch creates tensors using 32-bit floating-point numbers (float32), whereas NumPy used 64-bit numbers (float64). The lower precision accounts for the tiny discrepancy in the trailing digits. If exact agreement were needed, we could create the tensor with dtype=torch.float64, but we leave PyTorch's default in place for simplicity. The more important point is *how* the gradient was obtained: the NumPy version approximated each partial derivative with two extra function evaluations per parameter, while the PyTorch version computed all partial derivatives in a single backward pass.

For a two-parameter toy problem this distinction is academic, but it becomes decisive at scale. When we later train neural network policies with millions of parameters, computing gradients by finite differences would require millions of perturbed function evaluations per step, which is utterly impractical. Automatic differentiation computes the gradient with respect to *all* parameters in roughly the cost of a single forward pass, regardless of how many parameters there are. This efficiency, combined with the exactness of the derivatives, is why frameworks like PyTorch are the foundation of modern deep reinforcement learning.

Note also that we deliberately wrote our own update loop here to keep the parallel with the NumPy version transparent. In practice, PyTorch provides torch.optim optimizers (such as SGD or Adam) that encapsulate this update logic, the gradient zeroing, and adaptive learning rate schemes. We will adopt those tools once we move on to real reinforcement learning tasks; for now, writing the loop by hand makes the underlying mechanics easy to see.

This basic understanding of gradients and gradient ascent is fundamental to many machine learning algorithms, including the policy gradient methods we will discuss next.