Week 2 Optimisation for Machine Learning

Neimhin Robinson Gunning, 16321701

March 12, 2024

Let

$$f(x, y) = 3(x-5)^4 + 10(y-9)^2 \tag{1}$$

and

$$g(x, y) = \max(x-5, 0) + 10|y-9| \tag{2}$$

Using `sympy` we find the derivatives:

$$\nabla f = [\frac{df}{dx}, \frac{df}{dy}] = [12(x-5)^3, 20y - 180]$$

$$\nabla g = [\frac{dg}{dx}, \frac{dg}{dy}] = [\text{Heaviside}(x-5), 10\text{sign}(y-9)]$$

Clearly, the minimum of $f(x, y)$ is 0 and they is minimized by $x = 5$, $y = 9$. The other function $g(x, y)$ also has minimum 0 but is minized by any of $x \in [-\infty, 5]$ and $y = 9$.

1 (a)

1.1 (a) (i) Polyak

The Polyak step size is

$$\alpha_{\text{Polyak}} = \frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x)} \tag{3}$$

where x is the parameter vector, $f(x)$ is the function to optimise, and $f^* \approx \min_x f(x)$.

funcs.txt   Wed Feb 21 15:03:56 2024   1

```
function: 3*(x-5)^4+10*(y-9)^2
function: Max(x-5,0)+10*|y-9|
```

Figure 1: Two bivariate functions downloaded from `https://www.scss.tcd.ie/Doug.Leith/CS7DS2/week4.php`

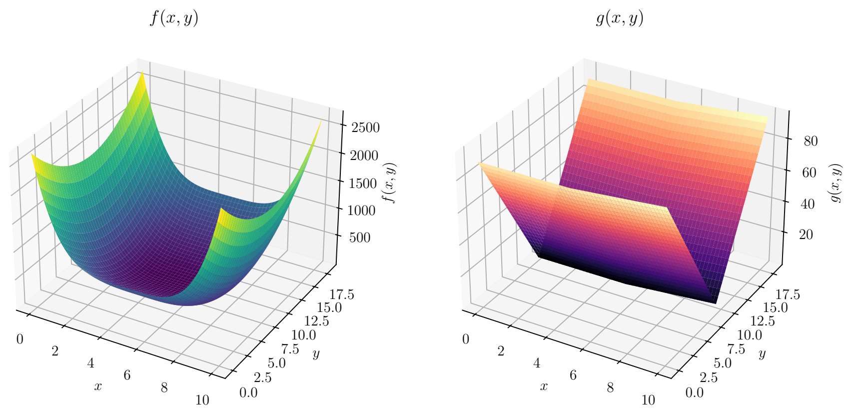

Figure 2

Gradient descent iteration with Polyak step size is implemented in Listing 1. The function is evaluated at the current value for x and the numerator is calculated: $f(x) - f^*$. A reasonable estimate for the minimum of the function, f^*, is required, here assumed to be 0. The dot product of the gradient is taken as the denominator. The step size is $\frac{f(x)-f^*}{\nabla f(x)^T \nabla f(x)}$. We multiply the step size by the gradient and subtract the result from the current x.

Listing 1: An implementation of the update step of gradient descent using Polyak step size.

src/polyak.py **Tue Mar 12 15:12:27 2024** **1**

```
import numpy as np

def iterate(self):
    self._x_value = self._start
    self._old_x_value = None
    self._f_star = 0
    self._iteration = 0
    self._converged_value = False
    self._grad_value = self._gradient(self._x_value)

    yield self.state_dict()

    while not self._converged_value:
        if self._max_iter > 0 and self._iteration > self._max_iter:
            break
        numerator = self._function(self._x_value) - self._f_star
        self._grad_value = self._gradient(self._x_value)
        denominator = np.dot(self._grad_value, self._grad_value) # sum of element-wise products
        self._old_x_value = self._x_value
        step = numerator/denominator
        self._x_value = self._x_value - step * self._grad_value
        self._converged_value = self._converged(self._x_value, self._old_x_value)
        yield self.state_dict()
```

1.2 (a) (ii) RMSProp

The RMSProp step size at iteration t is

$$\alpha_t = \frac{\alpha_0}{\epsilon + \sqrt{(1-\beta)\sum_{i=0}^{t-1} \beta^{t-i} (\nabla f(x_i))^2}} \tag{4}$$

and the update rule is

$$x_{t+1} := x_t - \alpha_t * \nabla f(x_t) \tag{5}$$

where ϵ is some small value to prevent divide by zero, α_0 and β are hyperparameters to be set, noting that $0 < \beta \leq 1$. The result is that previous gradients influence the current step size, but are gradually forgotten due to the β^{t-i} term.

A Python implementation of the update step is provided in Listing 2. The term inside the square root can be calculated iteratively, as in line 25 of Listing 2.

Listing 2: An implementation of the update step of gradient descent using RMSProp step size.

```
src/rmsprop.py        Tue Mar 12 15:12:45 2024        1
 1: def iterate(self):
 2:     import numpy as np
 3:     self._x_value = self._start
 4:     old_x_value = None
 5:     self._iteration = 0
 6:     self._sum = np.zeros(self._x_value.shape)
 7:     alpha_n = np.zeros(self._x_value.shape)
 8:     alpha_n.fill(self._step_size)
 9:     self._converged_value = False
10:     self._grad_value = self._gradient(self._x_value)
11:
12:     yield self.state_dict()
13:
14:     while not self._converged_value:
15:         self._iteration += 1
16:         if self._max_iter > 0 and iteration > self._max_iter:
17:             break
18:         self._grad_value = self._gradient(self._x_value)
19:         old_x_value = self._x_value
20:         self._x_value = self._x_value - alpha_n * self._grad_value
21:         self._sum = self._beta * self._sum + (1-self._beta) * (self._grad_value**2)
22:         alpha_n = self._step_size / (self._sum**0.5+self._epsilon)
23:         self._converged_value = self._converged(self._x_value, old_x_value)
24:         yield self.state_dict()
```

1.3 (a) (iii) Heavy Ball

The Heavy Ball step is

$$z_{t+1} = \beta z_t + \alpha \nabla f(x_t) \tag{6}$$

with the update rule

$$x_{t+1} = x_t - z_{t+1} \tag{7}$$

where t is the current iteration (starting at 0), $z_0 = 0$, and x_0, α, and β have to be set.

A Python implementation of the update step is provided in Listing 3.

Listing 3: An implementation of the update step of gradient descent using Heavy Ball step size.

src/heavy_ball.py **Tue Mar 12 14:57:31 2024** **1**

```python
import lib


def iterate(self):
    self._x_value = self._start
    self._old_x_value = None
    self._iteration = 0
    self._converged_val = False
    self._grad_value = self._gradient(self._x_value)
    self._z = 0
    yield self.state_dict()  # yield initial values

    while not self._converged_val:
        self._iteration += 1
        if self._max_iter > 0 and self._iteration > self._max_iter:
            break
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._z = self._beta * self._z + self._step_size * self._grad_value
        self._x_value = self._x_value - self._z
        self._converged_val = self._converged(self._x_value, self._old_x_value)
        yield self.state_dict()
```

1.4 (a) (iv) Adam

The Adam step size is calculated in terms of

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1)\nabla f(x_t) \tag{8}$$

and

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)[\nabla f(x_t) \circ \nabla f(x_t)] \tag{9}$$

from which we get

$$\hat{m} = \frac{m_{t+1}}{(1 - \beta_1^t)} \tag{10}$$

and

$$\hat{v} = \frac{m_{t+1}}{(1 - \beta_2^t)} \tag{11}$$

which are used in the update step as

$$x_{t+1} = x_t - \alpha[\frac{\hat{m}_1}{\epsilon + \sqrt{\hat{v}_1}}, \ldots, \frac{\hat{m}_n}{\epsilon + \sqrt{\hat{v}_n}}] \tag{12}$$

where t is the iteration, α, β_1, and β_2 are hyperparameters, and ϵ is some small value to prevent divide-by-zero.

A Python implementation of the update step is provided in Listing 4.

Listing 4: An implementation of the update step of gradient descent using Adam step size.

src/adam.py Tue Mar 12 14:57:31 2024 1

```python
import lib
import numpy as np
import json


def iterate(self):
    self._x_value = self._start
    self._old_x_value = None
    self._iteration = 0
    self._m = np.zeros(self._x_value.shape, dtype=np.float64)
    self._v = np.zeros(self._x_value.shape, dtype=np.float64)
    self._converged_value = False
    self._grad_value = self._gradient(self._x_value)

    yield self.state_dict()

    while not self._converged_value:
        if self._max_iter > 0 and self._iteration > self._max_iter:
            break
        self._grad_value = self._gradient(self._x_value)
        self._m = self._beta * self._m + (1-self._beta)*self._grad_value
        # grad_value * grad_value gives element-wise product of np array
        self._v = self._beta2 * self._v + (1-self._beta2) * (self._grad_value*self._grad_value)
        self._old_x_value = self._x_value
        self._iteration += 1
        m_hat = self._m / (1-(self._beta ** self._iteration))
        v_hat = np.array(self._v / (1-(self._beta2 ** self._iteration)))
        v_hat_aug = v_hat**(0.5) + self._epsilon
        adam_grad = m_hat / v_hat_aug
        self._x_value = self._x_value - self._step_size * adam_grad
        self._converged_value = self._converged(self._x_value, self._old_x_value)
        yield self.state_dict()
```