6-1_Adam

April 14, 2021

1 Adam

In this exercise you'll implement Adam from scratch by adding code to the provided functions. All places that need adjustments are started with a comment and todo: giving instructions.

[ ]:
```
%matplotlib notebook
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib
import numpy as np

# Parameters
x_min = -5
x_max = 5
y_min = -5
y_max = 5
```

We are going to start by defining a loss function

[ ]:
```
def function(x):
    return (x[0] ** 2 + x[1] - 11) ** 2 + (x[0] + x[1] ** 2 - 7) ** 2


# Todo:
# fill function grad - The gradient of function.
# Note, the current function body is just there to illustrate which output is␣
 ↪expected for futher code.
# As for the lecture the gradients *-1 are plotted over the contour plot of the␣
 ↪loss landscape in the next cell.
# You can use this to "test" your grad function

def grad(x):
    return [0,0]
```

[ ]:
```
# Plotting
fig = plt.figure()
ax = fig.gca()
```

```
x = np.linspace(x_min, x_max, 100)
y = np.linspace(y_min, y_max, 100)
X, Y = np.meshgrid(x, y)
Z = function(x=[X, Y])
levels = np.logspace(-1.5, 4, 30)
cset = ax.contour(
    X, Y, Z, levels=levels, cmap=cm.coolwarm, norm=matplotlib.colors.LogNorm()
)
plt.colorbar(cset)
ax.set_xlabel("$x_0$")
ax.set_xlim(x_min, x_max)
ax.set_ylabel("$x_1$")
ax.set_ylim(y_min, y_max)

x_quiver = np.linspace(x_min, x_max, 15)
y_quiver = np.linspace(y_min, y_max, 15)
X_quiver, Y_quiver = np.meshgrid(x_quiver, y_quiver)
U = np.zeros(shape=X_quiver.shape)
V = np.zeros(shape=X_quiver.shape)

for ii in range(U.shape[0]):
    for jj in range(U.shape[1]):
        U[ii, jj] = -1 * grad([X_quiver[ii, jj], Y_quiver[ii, jj]])[0]
        V[ii, jj] = -1 * grad([X_quiver[ii, jj], Y_quiver[ii, jj]])[1]

q = ax.quiver(X_quiver, Y_quiver, U, V)


plt.show()
```

1.1 Implementing the optimizer

[ ]:
```
class Adam:
    def __init__(self, theta_0, grad_fc, beta1=0.9, beta2=0.999, alpha=0.1):
        self.theta = theta_0

        self.beta1 = beta1
        self.beta2 = beta2
        self.alpha = alpha
        self.epsilon = 1e-8

        self.m_t_1 = np.zeros([1, len(theta_0)])
        self.v_t_1 = np.zeros([1, len(theta_0)])
        self.t = 0

        self.grad = grad_fc
```

```
    def step(self):
        # Todo: implement the gradient step for Adam use the class attributes
        # Note that you can use the two cells below to run your optimizer as␣
↪long as you don't change the interfaces

        return self.theta
```

[ ]:
```
theta_0 = np.random.uniform([-2, -2], [[1, 1]]).squeeze()
theta_log = [list(theta_0)]
optimizier = Adam(theta_0, grad, alpha=0.1)

for ii in range(100):
    theta = optimizier.step()
    theta_log.append(list(theta))
theta_log = np.array(theta_log)
```

[ ]:
```
# Plotting
fig = plt.figure()
ax = fig.gca()

x = np.linspace(x_min, x_max, 100)
y = np.linspace(y_min, y_max, 100)
X, Y = np.meshgrid(x, y)
Z = function(x=[X, Y])
levels = np.logspace(-1.5, 4, 30)
cset = ax.contour(
    X, Y, Z, levels=levels, cmap=cm.coolwarm, norm=matplotlib.colors.LogNorm()
)
plt.colorbar(cset)

ax.set_xlabel("$x_0$")
ax.set_xlim(x_min, x_max)
ax.set_ylabel("$x_1$")
ax.set_ylim(y_min, y_max)

ax.plot(theta_log[:, 0], theta_log[:, 1], "k.-", label="Adam")
plt.legend()
plt.show()
```

1.2 Ideas for next steps

If you're done with this exercise you might go ahead and try out extensions or additions like: * Implement SGD and momentum and compare them with Adam * Explore the trade off between learning rate and momentum factors * Implement a different loss function and respective gradient * Fix the inital guess and tune your hyperparamters parameters to reach a local optimum with as few steps as possible