Optimisation Basics 3 — Coding Practice Questions

CM52054: Foundational Machine Learning
Practice set with fully worked answers

Newton's Method

1) **Newton's method for linear regression.**
Generate a synthetic 1D dataset from the model

$$y = 3x + 2 + \text{noise},$$

For a squared-error function $g(w) = \|Xw - y\|^2$, implement Newton's method:

$$w_{t+1} = w_t - [\nabla^2 g(w_t)]^{-1} \nabla g(w_t),$$

where

$$\nabla g(w) = 2X^\top (Xw - y), \qquad \nabla^2 g(w) = 2X^\top X.$$

Start from any initial guess and show that one Newton step recovers the closed-form solution up to numerical precision.

Answer:

```python
import numpy as np

# ----------------------------------------------------------
# 0. Generate data
# ----------------------------------------------------------
# Set a random seed so results are reproducible
np.random.seed(42)

# Number of data points
N = 100

# Generate x values uniformly in [0, 10]
x = np.linspace(0, 10, N)

# True underlying relationship: y = 3x + 2 + noise
true_w0 = 2.0 # intercept (bias term)
true_w1 = 3.0 # slope

# Add Gaussian noise with mean 0 and standard deviation 2
noise = np.random.normal(loc=0.0, scale=2.0, size=N)

# Generate noisy observations
y = true_w0 + true_w1 * x + noise

# We construct X as:
# X[:, 0] = 1 (bias / intercept column)
# X[:, 1] = x (feature values)
# So X has shape (N, 2).
X = np.column_stack((np.ones_like(x), x))
```

```python
# ----------------------------------------------------------
# 1. Define gradient and Hessian of g(w) = ||Xw - y||^2
# ----------------------------------------------------------

def grad_g(w, X, y):
    """
    Gradient of g(w) = ||Xw - y||^2:
        \delta g(w) = 2 X^T (Xw - y)
    """
    residual = X @ w - y
    return 2 * (X.T @ residual)

def hessian_g(X):
    """
    Hessian of g(w) = ||Xw - y||^2:
        \delta^2 g(w) = 2 X^T X (constant, independent of w)
    """
    return 2 * (X.T @ X)

# ----------------------------------------------------------
# 2. Perform a single Newton step
# ----------------------------------------------------------

# Initial guess (arbitrary)
w_init = np.array([0.0, 0.0])

# Compute gradient at initial point
g = grad_g(w_init, X, y)

# Compute Hessian (constant)
H = hessian_g(X)

# Invert Hessian
H_inv = np.linalg.inv(H)

# Newton update:
# w_new = w_init - H_inv * g
w_newton = w_init - H_inv @ g

print("After one Newton step:")
print(f"w0 = {w_newton[0]:.6f}, w1 = {w_newton[1]:.6f}")

# ----------------------------------------------------------
# 3. Compare to the normal equation solution
# ----------------------------------------------------------

w_closed = np.linalg.inv(X.T @ X) @ (X.T @ y)

print("Closed-form solution:")
print(f"w0 = {w_closed[0]:.6f}, w1 = {w_closed[1]:.6f}")

# Norm of the difference should be extremely small
diff = np.linalg.norm(w_newton - w_closed)
print(f"Difference between Newton and closed-form: {diff:.4e}")
```

2) **Visualising gradient descent vs Newton on an anisotropic function.**

Consider the 2D linear function

$$f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^\top A\mathbf{x}, \quad A = \begin{bmatrix} 5 & 0 \\ 0 & 0.5 \end{bmatrix}.$$

Implement both gradient descent and Newton's method starting from $\mathbf{x}_0 = (6, 6)$ and plot their trajectories on the same contour plot to show that gradient descent zig-zags while Newton's method moves directly to the minimum.

Answer:

```
import numpy as np
import matplotlib.pyplot as plt

# ------------------------------------------------------
# 1. Define the quadratic function and its derivatives
# ------------------------------------------------------

# Matrix A that defines curvature (anisotropic: one direction steep, one flat)
A = np.array([[5.0, 0.0],
              [0.0, 0.5]])

def f(x):
    """
    Quadratic function:
        f(x) = 0.5 * x^T A x
    where x is a 2D vector.
    """
    return 0.5 * x.T @ A @ x

def grad_f(x):
    """
    Gradient of f(x):
        \delta f(x) = A x
    """
    return A @ x

def hessian_f():
    """
    Hessian of f(x) is simply A, and it is constant.
    """
    return A

# ------------------------------------------------------
# 2. Implement Gradient Descent and Newton's Method
# ------------------------------------------------------

# Starting point (far from optimum at [0, 0])
x0 = np.array([6.0, 6.0])

alpha = 0.25 # step size for gradient descent
num_steps = 8 # number of update steps to visualise

# Store trajectory points for each method
gd_points = [x0.copy()]
newton_points = [x0.copy()]

x_gd = x0.copy()
x_newton = x0.copy()
```

```
H = hessian_f()
H_inv = np.linalg.inv(H)

for _ in range(num_steps):
    # ----- Gradient Descent -----
    g = grad_f(x_gd)
    x_gd = x_gd - alpha * g
    gd_points.append(x_gd.copy())

    # ----- Newton's Method -----
    g_newton = grad_f(x_newton)
    x_newton = x_newton - H_inv @ g_newton
    newton_points.append(x_newton.copy())

gd_points = np.array(gd_points)
newton_points = np.array(newton_points)

# ---------------------------------------------------------
# 3. Plot contour lines and both trajectories
# ---------------------------------------------------------

# Create a grid on which to evaluate f(x)
xx = np.linspace(-7, 7, 200)
yy = np.linspace(-7, 7, 200)
XX, YY = np.meshgrid(xx, yy)

# Compute function values on grid
ZZ = 0.5 * (A[0, 0] * XX**2 + A[1, 1] * YY**2)

plt.figure(figsize=(7, 7))

# Draw level sets (contours) of the function
contours = plt.contour(XX, YY, ZZ, levels=15)
plt.clabel(contours, inline=True, fontsize=8)

# Plot gradient descent path (orange circles)
plt.plot(gd_points[:, 0], gd_points[:, 1],
        marker='o', color='orange', label="Gradient Descent")

# Plot Newton's method path (green stars)
plt.plot(newton_points[:, 0], newton_points[:, 1],
        marker='*', color='green', label="Newton's Method")

# Mark the optimum at (0, 0)
plt.scatter(0, 0, marker='*', s=200, color='red', label="Optimum")

plt.title("Gradient Descent vs Newton's Method")
plt.xlabel("x1")
plt.ylabel("x2")
plt.legend()
plt.grid(True)
plt.axis('equal') # keep aspect ratio so ellipses are not distorted
plt.show()
```

3) **Optional: Numerical gradient & Hessian checker.**

For the function

$$f(x, y) = x^3 + xy^2,$$

(a) implement analytical gradient and Hessian;
(b) implement numerical approximations of gradient and Hessian using central finite differences;
(c) evaluate both at a few random points and print the differences to verify that your analytical derivatives are correct.

Analytical derivatives:

$$\frac{\partial f}{\partial x} = 3x^2 + y^2, \quad \frac{\partial f}{\partial y} = 2xy,$$

$$\nabla^2 f = \begin{bmatrix} 6x & 2y \\ 2y & 2x \end{bmatrix}.$$

Answer:

```python
import numpy as np

# ----------------------------------------------------------
# 1. Define function, gradient and Hessian analytically
# ----------------------------------------------------------

def f_xy(x):
    """
    Scalar function of 2 variables:
        f(x, y) = x^3 + x y^2

    Input:
        x : np.array([x1, x2]) where x1 = x, x2 = y
    """
    x1, x2 = x
    return x1**3 + x1 * x2**2

def grad_f_xy(x):
    """
    Analytical gradient:
        df/dx = 3x^2 + y^2
        df/dy = 2xy
    """
    x1, x2 = x
    dfdx = 3 * x1**2 + x2**2
    dfdy = 2 * x1 * x2
    return np.array([dfdx, dfdy])

def hessian_f_xy(x):
    """
    Analytical Hessian:
        d2f/dx2 = 6x
        d2f/dy2 = 2x
        d2f/dxdy = d2f/dydx = 2y
    """
    x1, x2 = x
    d2fdx2 = 6 * x1
    d2fdy2 = 2 * x1
    d2fdxdy = 2 * x2
    return np.array([[d2fdx2, d2fdxdy],
                     [d2fdxdy, d2fdy2]])
```

```
# ------------------------------------------------------
# 2. Numerical gradient and Hessian via finite differences
# ------------------------------------------------------

def numerical_grad(f, x, h=1e-5):
    """
    Numerical gradient using central differences.

    For each dimension i:
        f/x_i  ( f(x + h e_i) - f(x - h e_i) ) / (2h)
    where e_i is the i-th standard basis vector.
    """
    x = x.astype(float)
    grad = np.zeros_like(x)
    for i in range(len(x)):
        e = np.zeros_like(x)
        e[i] = 1.0
        grad[i] = (f(x + h * e) - f(x - h * e)) / (2 * h)
    return grad

def numerical_hessian(f, x, h=1e-4):
    """
    Numerical Hessian using central differences.

    For each pair (i, j):
        ^2 f / x_i x_j
        ( f(x + h e_i + h e_j) - f(x + h e_i - h e_j)
          - f(x - h e_i + h e_j) + f(x - h e_i - h e_j) ) / (4 h^2)
    """
    x = x.astype(float)
    n = len(x)
    H = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            e_i = np.zeros_like(x)
            e_j = np.zeros_like(x)
            e_i[i] = 1.0
            e_j[j] = 1.0
            f_pp = f(x + h*e_i + h*e_j)
            f_pm = f(x + h*e_i - h*e_j)
            f_mp = f(x - h*e_i + h*e_j)
            f_mm = f(x - h*e_i - h*e_j)
            H[i, j] = (f_pp - f_pm - f_mp + f_mm) / (4 * h**2)
    return H

# ------------------------------------------------------
# 3. Compare analytical vs numerical derivatives
# ------------------------------------------------------

np.random.seed(0)

for k in range(3):
    # Random test point in [-2, 2]^2
    x_test = np.random.uniform(-2, 2, size=2)

    g_analytic = grad_f_xy(x_test)
    g_numeric = numerical_grad(f_xy, x_test)
```

```
    H_analytic = hessian_f_xy(x_test)
    H_numeric = numerical_hessian(f_xy, x_test)

    print("\nTest point {k+1}: x = {x_test}")
    print("Gradient (analytic):", g_analytic)
    print("Gradient (numeric) :", g_numeric)
    print("Gradient diff norm :", np.linalg.norm(g_analytic - g_numeric))

    print("\nHessian (analytic):\n", H_analytic)
    print("Hessian (numeric):\n", H_numeric)
    print("Hessian diff norm :", np.linalg.norm(H_analytic - H_numeric))
```