Optimisation Basics 2 — Coding Practice Questions

CM52054: Foundational Machine Learning
Practice set with fully worked answers

Linear Regression & Gradient Descent

1) **1D linear regression: closed form vs gradient descent.**

   Task:

   a) Generate synthetic 1D data from the model $y = 3x + 2 + \varepsilon$ with Gaussian noise.

   b) Fit a linear model $f(x) = w_1 x + w_0$ using:

      i. the normal equation (closed form), and

      ii. batch gradient descent on the squared loss.

   c) Compare the learned parameters to the true ones.

   Answer:

```python
import numpy as np

# ------------------------------
# 1. Generate synthetic data
# ------------------------------

# Set random seed so that the results are reproducible
np.random.seed(42)

# Number of data points
N = 100

# Generate N input points uniformly in [0, 1]
x = np.random.rand(N)

# True underlying parameters for the data-generating process:
# y = true_w1 * x + true_w0 + noise
true_w1 = 3.0 # slope of the true line
true_w0 = 2.0 # intercept of the true line

# Additive Gaussian noise: mean = 0, standard deviation = 0.3
noise = 0.3 * np.random.randn(N)

# Output values follow the linear relation plus noise
y = true_w1 * x + true_w0 + noise

# ------------------------------
# 2. Build the matrix X
# ------------------------------
# We want to fit y = w0 + w1 * x.
# In matrix form, we write y = X w, where
# w = [w0, w1]^T
# X[i, :] = [1, x_i]
# The first column of X is all ones (for the intercept),
# the second column is the input x values.

X = np.column_stack([np.ones_like(x), x]) # shape: (N, 2)

# -----------------------------
# 3. Closed-form solution
# -----------------------------
# Normal equation:
# w* = (X^T X)^(-1) X^T y

XT = X.T # transpose of X
XTX = XT @ X # X^T X (2x2 matrix)
XTy = XT @ y # X^T y (2D vector)

# Compute inverse of X^T X and multiply by X^T y
w_closed_form = np.linalg.inv(XTX) @ XTy

# -----------------------------
# 4. Batch Gradient Descent
# -----------------------------

# Random initial guess for parameters w = [w0, w1]
w_gd = np.random.randn(2)

# Learning rate (step size) controls how big each update is
alpha = 0.1

# Number of gradient descent iterations
num_iters = 200

# Track the loss to see if it decreases over time (optional)
loss_history = []

for t in range(num_iters):
    # Predicted outputs for all N samples: y_pred = X w
    y_pred = X @ w_gd

    # Residuals (errors) for all samples: r_i = y_pred_i - y_i
    residuals = y_pred - y

    # Mean squared error (up to constant factor 1/2)
    # L(w) = (1/N) * sum_i r_i^2
    loss = np.mean(residuals**2)
    loss_history.append(loss)

    # Gradient of L(w) for linear regression with MSE:
    # \nabla_w L = (2/N) X^T (Xw - y)
    grad = (2.0 / N) * (X.T @ residuals)

    # Gradient descent update rule:
    # w_{t+1} = w_t - alpha * \nabla_w L
    w_gd = w_gd - alpha * grad

# -----------------------------
# 5. Print and compare results
# ------------------------------
print("True parameters: w0 = {:.3f}, w1 = {:.3f}".format(true_w0, true_w1))
print("Closed-form solution: w0 = {:.3f}, w1 = {:.3f}".format(w_closed_form[0],
    w_closed_form[1]))
print("Gradient descent: w0 = {:.3f}, w1 = {:.3f}".format(w_gd[0], w_gd[1]))
```

Explanation: We (i) construct the matrix X to include the intercept, (ii) compute the closed-form solution via the normal equation, and (iii) implement batch gradient descent using the analytic gradient of the MSE. Both methods should recover parameters close to the true line $y = 3x + 2$ up to noise.

2) **Batch GD vs SGD vs mini-batch GD.**

Task:

a) Reuse the synthetic linear regression data from Question 1.

b) Implement:

i. Batch gradient descent.

ii. Stochastic gradient descent (SGD).

iii. Mini-batch gradient descent.

c) Track the mean squared error over iterations for each method and compare:
- stability of the loss curve,
- speed of convergence.

Answer:

```
import numpy as np

np.random.seed(42)

# --------------------------------
# 1. Generate / reuse data as in Q1
# --------------------------------
N = 100
x = np.random.rand(N)

true_w1, true_w0 = 3.0, 2.0
noise = 0.3 * np.random.randn(N)
y = true_w1 * x + true_w0 + noise

# Matrix with intercept column
X = np.column_stack([np.ones_like(x), x]) # shape: (N, 2)

# Helper function to compute mean squared error
def mse_loss(X, y, w):
    """
    Compute mean squared error (MSE):
        L(w) = (1/N) * ||Xw - y||^2
    """
    residuals = X @ w - y
    return np.mean(residuals**2)

# General settings
```

```
alpha = 0.1 # learning rate
num_iters = 200 # number of iterations

# -----------------------------
# 2. Batch Gradient Descent
# -----------------------------
w_batch = np.random.randn(2) # random initial guess
loss_hist_batch = []

for t in range(num_iters):
    # Use ALL samples to compute gradient
    residuals = X @ w_batch - y

    # Gradient over the full dataset:
    # \nabla_w L = (2/N) X^T (Xw - y)
    grad = (2.0 / N) * (X.T @ residuals)

    # Batch GD update
    w_batch = w_batch - alpha * grad

    # Store the full-dataset loss
    loss_hist_batch.append(mse_loss(X, y, w_batch))

# -----------------------------
# 3. Stochastic Gradient Descent (SGD)
# -----------------------------
w_sgd = np.random.randn(2) # new random initial guess
loss_hist_sgd = []

for t in range(num_iters):
    # Randomly sample a single index i from {0, ..., N-1}
    i = np.random.randint(0, N)

    # Extract a single sample (Xi, yi)
    Xi = X[i:i+1, :] # keeps shape (1, 2)
    yi = y[i:i+1] # shape (1,)

    # Residual using only this single sample
    residual_i = Xi @ w_sgd - yi

    # SGD gradient using ONE sample:
    # gradient ~ 2 * Xi^T (Xi w - yi)
    grad_i = 2.0 * (Xi.T @ residual_i)

    # SGD update step
    w_sgd = w_sgd - alpha * grad_i.flatten()

    # Store full-dataset loss (for monitoring)
    loss_hist_sgd.append(mse_loss(X, y, w_sgd))

# -----------------------------
# 4. Mini-batch Gradient Descent
# -----------------------------
w_mb = np.random.randn(2)
loss_hist_mb = []

batch_size = 16 # typical mini-batch size
```

```
for t in range(num_iters):
    # Randomly choose 'batch_size' indices without replacement
    batch_indices = np.random.choice(N, batch_size, replace=False)

    # Form the mini-batch
    Xb = X[batch_indices]
    yb = y[batch_indices]

    # Residuals on the mini-batch
    residuals_b = Xb @ w_mb - yb

    # Gradient averaged over the mini-batch:
    # (2/B) Xb^T (Xb w - yb)
    grad_b = (2.0 / batch_size) * (Xb.T @ residuals_b)

    # Mini-batch GD update
    w_mb = w_mb - alpha * grad_b

    # Track full-dataset loss
    loss_hist_mb.append(mse_loss(X, y, w_mb))

# -----------------------------
# 5. Compare final parameters
# -----------------------------
print("True: w0 = {:.3f}, w1 = {:.3f}".format(true_w0, true_w1))
print("Batch GD: w0 = {:.3f}, w1 = {:.3f}".format(w_batch[0], w_batch[1]))
print("SGD: w0 = {:.3f}, w1 = {:.3f}".format(w_sgd[0], w_sgd[1]))
print("Mini-batch (16): w0 = {:.3f}, w1 = {:.3f}".format(w_mb[0], w_mb[1]))
```

Explanation: Batch GD uses the whole dataset in each update (stable but expensive per step), SGD uses a single random sample (fast but noisy), and mini-batch sits in between. Tracking MSE over iterations for each method illustrates the stability vs speed trade-off discussed in the slides.