Exercise of chapter 4

---

1. Which Linear Regression training algorithm can you use if you have a training set with millions of features?

You should use Stochastic Gradient Descent (SGD) or Mini-batch Gradient Descent.

- These methods scale well to large feature sets and datasets because they do not require loading the entire data matrix into memory.
- In contrast, the Normal Equation and Batch Gradient Descent become computationally expensive or infeasible due to matrix operations with very high dimensionality.

---

2. Suppose the features in your training set have very different scales. Which algorithms might suffer from this, and how? What can you do about it?

Algorithms that use Gradient Descent (Linear Regression, Logistic Regression, Neural Nets, etc.) will suffer.

- The cost function contours become elongated, making convergence slow or unstable.
  Solution: Apply feature scaling like:
- Standardization: x' = (x - mean) / std
- Normalization: x' = (x - min) / (max - min)

---

3. Can Gradient Descent get stuck in a local minimum when training a Logistic Regression model?

No, it cannot.

- The cost function for Logistic Regression is convex, meaning it has one global minimum.
- Gradient Descent may be slow or oscillate, but it won’t get stuck in local minima.

---

4. Do all Gradient Descent algorithms lead to the same model, provided you let them run long enough?

- If the optimization problem is convex (such as Linear Regression or Logistic Regression), and assuming the learning rate is not too high, then all Gradient Descent algorithms will approach the global optimum and end up producing fairly similar models. However, unless you gradually reduce the learning rate, Stochastic GD and Mini-batch GD will never truly converge; instead, they will keep jumping back and forth around the global optimum. This means that even if you let them run for a very long time, these Gradient Descent algorithms will produce slightly different models.

---

5. Suppose you use Batch Gradient Descent and you plot the validation error at every epoch. If the validation error consistently goes up, what is likely going on? How can you fix this?

- If the validation error consistently goes up after every epoch, then one possibility is that the learning rate is too high and the algorithm is diverging. If the training error also goes up, then this is clearly the problem and you should reduce the learning rate. However, if the training error is not going up, then your model is overfitting the training set and you should stop training.

---

6. Is it a good idea to stop Mini-batch Gradient Descent immediately when the validation error goes up?

No, because Mini-batch Gradient Descent has noisy updates.

- Validation error may go up temporarily, even if the model is improving overall.
  Better approach:
- Use early stopping with patience (e.g., stop only after 10 consecutive increases).

---

7. Which Gradient Descent algorithm will reach the vicinity of the optimal solution the fastest? Which will actually converge? How can you make the others converge as well?

Stochastic Gradient Descent has the fastest training iteration since it considers only one training instance at a time, so it is generally the first to reach the vicinity of the global optimum (or Mini-batch GD with a very small mini-batch size). However, only Batch Gradient Descent will actually converge, given enough training time. As mentioned, Stochastic GD and Mini-batch GD will bounce around the optimum, unless you gradually reduce the learning rate.

---

8. Suppose you are using Polynomial Regression. You plot the learning curves and notice a large gap between training and validation error. What is happening?

This is a case of high variance (overfitting).

1. Reduce model complexity (e.g., lower polynomial degree)
2. Add regularization (e.g., Ridge Regression)
3. Increase training data

---

9. Suppose you are using Ridge Regression and the training and validation errors are almost equal and fairly high. Would you say that the model suffers from high bias or high variance? Should you increase the regularization hyperparameter α or reduce it?

This is a case of high bias (underfitting).

- Since errors are high and similar, the model is too simple.
  You should reduce the regularization parameter α, allowing the model more flexibility.

---

10. Why would you want to use:

a. Ridge Regression instead of Linear Regression?
To reduce overfitting by shrinking coefficients. Especially useful when features are correlated.

b. Lasso instead of Ridge?
To get a sparse model (i.e., feature selection). Lasso can eliminate irrelevant features by setting their coefficients to zero.

c. Elastic Net instead of Lasso?
When you want a balance:

- Handles correlated features better than Lasso

- Encourages sparsity like Lasso
  Combines L1 (Lasso) and L2 (Ridge) penalties

---

11. Suppose you want to classify pictures as outdoor/indoor and daytime/nighttime. Should you implement two Logistic Regression classifiers or one Softmax Regression classifier?

You should implement two Logistic Regression classifiers.

- These are two separate binary classification tasks
- Softmax is for multi-class classification with mutually exclusive classes (e.g., cat/dog/horse)

---

12. Implement Batch Gradient Descent with early stopping for Softmax Regression (without using Scikit-Learn)

```
import numpy as np
def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  # stability
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
def compute_loss(y, y_hat):
    m = y.shape[0]
    log_likelihood = -np.log(y_hat[range(m), y])
    return np.mean(log_likelihood)
def one_hot(y, num_classes):
    one_hot = np.zeros((len(y), num_classes))
    one_hot[np.arange(len(y)), y] = 1
    return one_hot
def softmax_regression(X, y, lr=0.1, epochs=500, patience=10):
    m, n = X.shape
    k = len(np.unique(y))  # num classes
    X_b = np.c_[np.ones((m, 1)), X]  # add bias
    theta = np.random.randn(n + 1, k)
    y_onehot = one_hot(y, k)
    best_loss = np.inf
    patience_counter = 0
    for epoch in range(epochs):
        logits = X_b.dot(theta)
        y_hat = softmax(logits)
        loss = compute_loss(y, y_hat)
        if loss < best_loss:
            best_loss = loss
            best_theta = theta.copy()
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print(f"Early stopping at epoch {epoch}")
                break
        grad = 1/m * X_b.T.dot(y_hat - y_onehot)
        theta -= lr * grad
    return best_theta
```

You can call this function with training data (X, y) and get the trained weights with early stopping.