

Lab: Optimisation

ACTL3143 & ACTL5111 Deep Learning for Actuaries

As you have learned, a neural network consists of a set of weights and biases, and the network learns by adjusting these values so that we minimise the network's loss. Mathematically, we aim to find the optimum weights and biases (w^*, b^*) :

$$(w^*, b^*) = \arg \min_{w, b} \mathcal{L}(\mathcal{D}, (w, b))$$

where \mathcal{D} denotes the training data set and $\mathcal{L}(\cdot, \cdot)$ is the user-defined loss function.

Gradient descent is the method through which we update the weights and biases. We introduce two types of gradient descent: **stochastic** and **batch**.

- **Stochastic** gradient descent updates the weights and biases once for each observation in the data set.
- **Batch** gradient descent updates the values repeatedly by averaging the gradients across all the observations.

Example: Batch Gradient Descent for Linear Regression

Notation:

- $\mathcal{L}(\mathcal{D}, (w, b))$ denotes the loss function.
- $\hat{y}(x_i)$ denotes the predicted value for the i th observation $x_i \in \mathbb{R}^{d_x}$, where d_x represents the dimension of the input.
- N denotes the batch size.

The true model:

$$w_{\text{True}} = \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix}, b_{\text{True}} = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix}. \quad (1)$$

Obtain target values y :

```
import numpy as np
X = np.array([[1, 2], [3, 1], [1, 1]])
true_w = np.array([[1.5], [1.5]])
true_b = np.array([[0.1], [0.1], [0.1]])
y = X @ true_w + true_b
print(X); print(y)
```

```
[[1 2]
 [3 1]
 [1 1]]
[[4.6]
 [6.1]
 [3.1]]
```

A batch with $N = 3$ and $d_x = 2$:

$$X = \begin{pmatrix} 1 & 2 \\ 3 & 1 \\ 1 & 1 \end{pmatrix}, y = \begin{pmatrix} 4.6 \\ 6.1 \\ 3.1 \end{pmatrix}. \quad (2)$$

Step 1: Write down $\mathcal{L}(\mathcal{D}, (w, b))$ and \hat{y}^*

$$\mathcal{L}(\mathcal{D}, (w, b)) = \frac{1}{N} \sum_{i=1}^N (\hat{y}(x_i) - y_i)^2 = \frac{1}{N} (\hat{y} - y)^\top (\hat{y} - y), \quad (3)$$

where

$$\hat{y}(x_i) = x_i w + b, \quad (4)$$

$$\hat{y} = Xw + b = \begin{pmatrix} \hat{y}(x_1) \\ \hat{y}(x_2) \\ \hat{y}(x_3) \end{pmatrix}. \quad (5)$$

Step 2: Derive $\frac{\partial \mathcal{L}}{\partial \hat{y}}$, $\frac{\partial \hat{y}}{\partial w}$, and $\frac{\partial \hat{y}}{\partial b}$

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{2}{N} \sum_{i=1}^N (\hat{y}(x_i) - y_i) = \frac{2}{N} (\hat{y} - y), \quad (6)$$

$$\frac{\partial \hat{y}}{\partial w} = X, \quad (7)$$

$$\frac{\partial \hat{y}}{\partial b} = I_{3 \times 1} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}. \quad (8)$$

Step 3: Derive $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w} = \frac{2}{N} \sum_{i=1}^N x_i^\top (\hat{y}(x_i) - y_i) = \frac{2}{N} X^\top (\hat{y} - y), \quad (9)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = \frac{2}{N} \sum_{i=1}^N (\hat{y}(x_i) - y_i) = \frac{2}{N} I^\top (\hat{y} - y). \quad (10)$$

Step 4: Initialise the weights and biases. Evaluate the gradients.

$$w = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, b = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad (11)$$

Subsequently,

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{2}{3} \underbrace{\begin{pmatrix} 1 & 3 & 1 \\ 2 & 1 & 1 \end{pmatrix}}_X \left[\underbrace{\begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix}}_{\hat{y}} - \underbrace{\begin{pmatrix} 4.6 \\ 6.1 \\ 3.1 \end{pmatrix}}_y \right] = \begin{pmatrix} -6.000 \\ -4.267 \end{pmatrix}, \quad (12)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{3} \underbrace{\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}}_I \left[\underbrace{\begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix}}_{\hat{y}} - \underbrace{\begin{pmatrix} 4.6 \\ 6.1 \\ 3.1 \end{pmatrix}}_y \right] = -3.200. \quad (13)$$

```

#number of rows == number of observations in the batch
N = X.shape[0]
w = np.array([[1], [1]])
b = np.array([[0] * N]).reshape(N, 1)

#Gradients
y_hat = X @ w + b
dw = 2/N * X.T @ (y_hat - y)
db = 2/N * np.sum(y_hat - y)
print(dw); print(db)

```

```

[[-6.          ]
 [-4.26666667]]
-3.1999999999999993

```

Step 5: Pick a learning rate η and update the weights and biases.

$$\eta = 0.1, \quad (14)$$

$$w = w - \eta \frac{\partial \mathcal{L}}{\partial w} = \begin{pmatrix} 1.600 \\ 1.427 \end{pmatrix}, \quad (15)$$

$$b = b - \eta \frac{\partial \mathcal{L}}{\partial b} = \begin{pmatrix} 0.320 \\ 0.320 \\ 0.320 \end{pmatrix} \quad (16)$$

```

#specify a learning rate to update
eta = 0.1
w = w - eta * dw
b = b - eta * db
print(w); print(b)

```

```

[[1.6          ]
 [1.42666667]]
[[0.32]
 [0.32]
 [0.32]]

```

Next Step: Update until convergence.

```

#loss function
def mse(y_pred, y_true):
    return(np.mean((y_pred-y_true)**2))

def lr_gradient_descent(eta = 0.1, w = None, b = None,
                        max_iter = 100, tol = 1e-08):
    """
    Gradient descent optimization for linear regression.

    Parameters:
    eta: float - learning rate (default=0.1)
    w: numpy array of shape (d, 1) - initial weights (default=ones)
    b: numpy array of shape (N, 1) - initial bias (default=zeros)
    max_iter: int - maximum number of iterations (default=100)
    tol: float - tolerance for stopping criteria (default=1e-08)

    Returns:
    w, b - optimized weights and bias
    """
    N, d = X.shape

    if w is None:
        w = np.ones((d, 1))
    if b is None:
        b = np.zeros((N, 1))

    prev_error = np.inf
    for _ in range(max_iter):
        y_hat = X @ w + b
        error = mse(y_hat, y)
        if np.abs(error - prev_error) < tol:
            return(w, b)
            break
        prev_error = error

        dw = 2/N * X.T @ (y_hat - y)
        db = 2/N * (y_hat - y)
        w -= eta * dw
        b -= eta * db
    return(w, b)

#Default initialisation

```

```
w_updated, b_updated = lr_gradient_descent(max_iter = 1000)
print(w_updated)
print(b_updated)
```

```
[[1.51250004]
 [1.47499994]]
[[0.13766113]
 [0.08758062]
 [0.11209706]]
```

Different Learning Rates and Initialisations

See more details in `Week_2_Tutorial_Notebook`.

Exercises

1. Apply stochastic gradient descent for the example given in Section 4.3.
2. Apply batch gradient descent for logistic regression. Follow the steps and information in Section 4.3.