

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 $(\mathbf{w}^*, \mathbf{b}^*)$:

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

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.
- **Mini-Batch** gradient descent updates the values repeatedly by averaging the gradients across a group of the observations (the 'mini-batch', or just 'batch').

Example: Mini-Batch Gradient Descent for Linear Regression

Notation:

- $\mathcal{L}(\mathcal{D}, (\mathbf{w}, \mathbf{b}))$ denotes the loss function.
- $\hat{y}(\mathbf{x}_i)$ denotes the predicted value for the i th observation $\mathbf{x}_i \in \mathbb{R}^{1 \times p}$, where p represents the dimension of the input.
- $\mathbf{w} \in \mathbb{R}^{p \times 1}$ denotes the weights.
- N denotes the batch size.

The model is

$$\hat{y}_i = \hat{g}(\mathbf{x}_i) = \mathbf{x}_i \mathbf{w} + b, \quad i = 1, \dots, n.$$

Let's set $p = 2$ and consider the true weights and bias as

$$\mathbf{w}_{\text{True}} = \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix}, b_{\text{True}} = 0.1.$$

Let's just make some toy dataset (batch) to train on:

```
import numpy as np

# Make up (arbitrarily) 12 observations with two features.
X = np.array([[1, 2],
               [3, 1],
               [1, 1],
               [0, 1],
               [2, 2],
               [-2, 3],
               [1, 2],
               [-1, -0.5],
               [0.5, 1.2],
               [2, 1],
               [-2, 3],
               [-1, 1]
              ])

w_true = np.array([[1.5], [1.5]])
b_true = 0.1

y = X @ w_true + b_true
print(X); print(y)
```

```
[[ 1.  2. ]
 [ 3.  1. ]
 [ 1.  1. ]
 [ 0.  1. ]
 [ 2.  2. ]
 [-2.  3. ]
 [ 1.  2. ]
 [-1. -0.5]
 [ 0.5 1.2]
 [ 2.  1. ]
 [-2.  3. ]
 [-1.  1. ]]
```

```

[-1.  -0.5]
[ 0.5  1.2]
[ 2.   1. ]
[-2.   3. ]
[-1.   1. ]]
[[ 4.6 ]
 [ 6.1 ]
 [ 3.1 ]
 [ 1.6 ]
 [ 6.1 ]
 [ 1.6 ]
 [ 4.6 ]
 [-2.15]
 [ 2.65]
 [ 4.6 ]
 [ 1.6 ]
 [ 0.1 ]]

```

If the batch size is $N = 3$, the first batch of observations is

$$\mathbf{X}_{1:3} = \begin{pmatrix} 1 & 2 \\ 3 & 1 \\ 1 & 1 \end{pmatrix}, \mathbf{y}_{1:3} = \begin{pmatrix} 4.6 \\ 6.1 \\ 3.1 \end{pmatrix}.$$

For simplicity, we will denote $\mathbf{X}_{1:3}$ as \mathbf{X} and $\mathbf{y}_{1:3}$ as \mathbf{y} .

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

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

where

$$\hat{y}(\mathbf{x}_i) = \mathbf{x}_i \mathbf{w} + b, \quad (2)$$

$$\hat{\mathbf{y}} = \mathbf{X} \mathbf{w} + b \mathbf{1} = \begin{pmatrix} \hat{y}(\mathbf{x}_1) \\ \hat{y}(\mathbf{x}_2) \\ \hat{y}(\mathbf{x}_3) \end{pmatrix}. \quad (3)$$

with $\mathbf{1}$ is a length 3 column vector of ones.

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

$$\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} = \frac{2}{N}(\hat{\mathbf{y}} - \mathbf{y}), \quad (4)$$

$$\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{w}} = \mathbf{X}, \quad (5)$$

$$\frac{\partial \hat{\mathbf{y}}}{\partial b} = \mathbf{1}. \quad (6)$$

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

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \left(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \right)^\top \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{w}} = \left(\frac{2}{N}(\hat{\mathbf{y}} - \mathbf{y}) \right)^\top \mathbf{X} = \frac{2}{N} \mathbf{X}^\top (\hat{\mathbf{y}} - \mathbf{y}), \quad (7)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \left(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \right)^\top \frac{\partial \hat{\mathbf{y}}}{\partial b} = \left(\frac{2}{N}(\hat{\mathbf{y}} - \mathbf{y}) \right)^\top \mathbf{1} = \frac{2}{N} \mathbf{1}^\top (\hat{\mathbf{y}} - \mathbf{y}). \quad (8)$$

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

$$\mathbf{w}^{(0)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, b^{(0)} = 0.$$

Subsequently,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} \Big|_{\mathbf{w}^{(0)}} = \frac{2}{3} \underbrace{\begin{pmatrix} 1 & 3 & 1 \\ 2 & 1 & 1 \end{pmatrix}}_{\mathbf{X}^\top} \left[\underbrace{\begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix}}_{\hat{\mathbf{y}}} - \underbrace{\begin{pmatrix} 4.6 \\ 6.1 \\ 3.1 \end{pmatrix}}_{\mathbf{y}} \right] = \begin{pmatrix} -6.000 \\ -4.267 \end{pmatrix}, \quad (9)$$

$$\frac{\partial \mathcal{L}}{\partial b} \Big|_{b^{(0)}} = \frac{2}{3} \underbrace{\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}}_{\mathbf{1}^\top} \left[\underbrace{\begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix}}_{\hat{\mathbf{y}}} - \underbrace{\begin{pmatrix} 4.6 \\ 6.1 \\ 3.1 \end{pmatrix}}_{\mathbf{y}} \right] = -3.200. \quad (10)$$

```
#number of rows == number of observations in the batch
X_batch = X[:3]
y_batch = y[:3]
```

```

N = X_batch.shape[0]
w = np.array([[1], [1]])
b = 0

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

```

```

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

```

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

$$\eta = 0.1, \quad (11)$$

$$\mathbf{w}^{(1)} = \mathbf{w}^{(0)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}} \Big|_{\mathbf{w}^{(0)}} = \begin{pmatrix} 1.600 \\ 1.427 \end{pmatrix}, \quad (12)$$

$$b^{(1)} = b^{(0)} - \eta \frac{\partial \mathcal{L}}{\partial b} \Big|_{b^{(0)}} = 0.320 \quad (13)$$

```

#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]]
0.31999999999999995

```

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(X, y, batch_size=32, eta=0.1, w=None, b=None, max_iter=100, tol=1e-08)
    """
    Gradient descent optimization for linear regression with random batch updates.

    Parameters:
    eta: float - learning rate (default=0.1)
    w: numpy array of shape (p, 1) - initial weights (default=ones)
    b: float - initial bias (default=zero)
    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, p = X.shape

    if w is None:
        w = np.ones((p, 1))
    if b is None:
        b = 0

    prev_error = np.inf
    batch_size = min(N, batch_size)
    num_batches = N//batch_size

    for iteration in range(max_iter):
        indices = np.arange(N)
        np.random.shuffle(indices)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        for batch in range(num_batches):
            start = batch * batch_size
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            y_hat = X_batch @ w + b
            error = mse(y_hat.squeeze(), y_batch.squeeze())

            if np.abs(error - prev_error) < tol:

```

```

        return w, b

    prev_error = error

    dw = 2 / batch_size * X_batch.T @ (y_hat - y_batch)
    db = 2 / batch_size * np.sum(y_hat - y_batch)

    w -= eta * dw
    b -= eta * db

    return w, b

#Default initialisation
w_updated, b_updated = lr_gradient_descent(X, y, batch_size = 3, max_iter = 1000)
print(w_updated)
print(b_updated)

```

```

[[1.49988083]
 [1.49951612]]
0.10090810316795538

```

Different Learning Rates and Initialisations

See more details in [maths-of-neural-networks.ipynb](#).

Exercises

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