Lab: Maths of Deep Learning

ACTL3143 & ACTL5111 Deep Learning for Actuaries

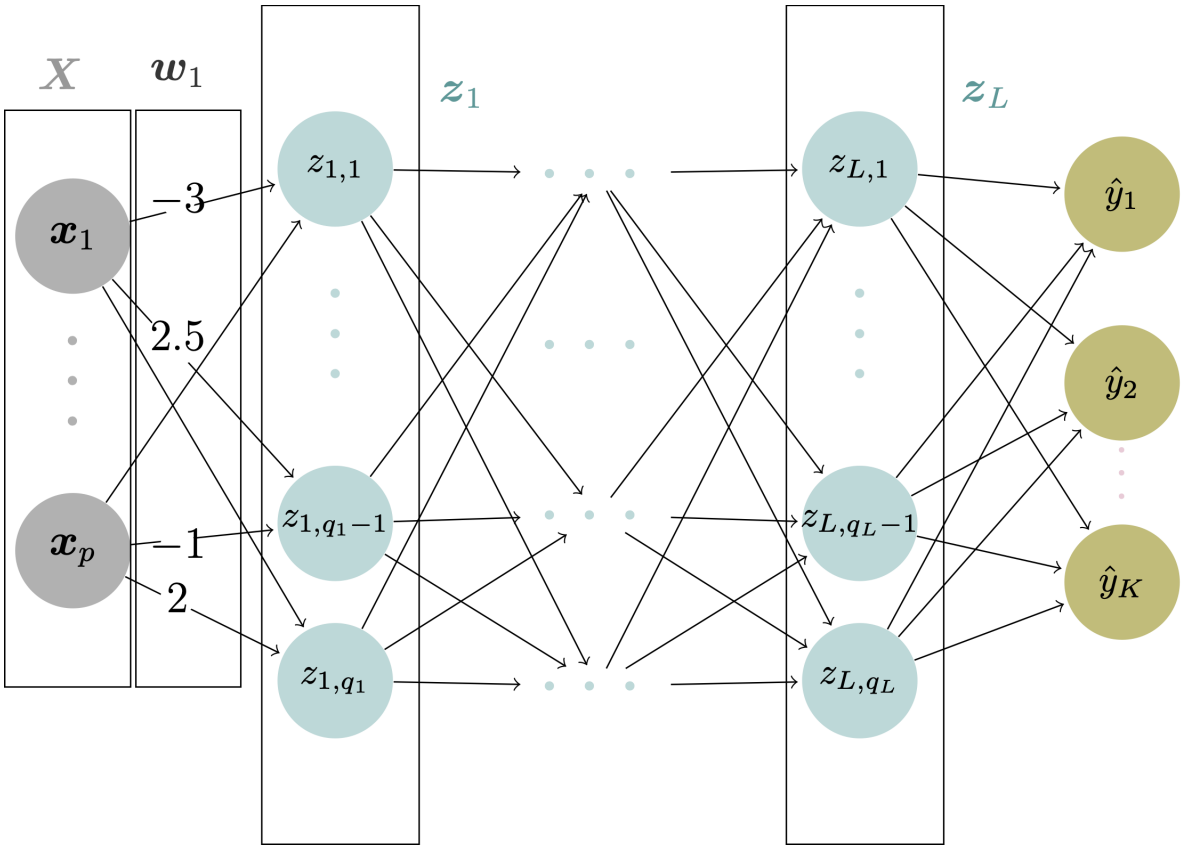


Figure 1: The structure of a neural network.

At each node in the hidden and output layers, the value $\boldsymbol{z}$ is calculated as a weighted sum of the node outputs in the previous layer, plus a bias. In other words:

$$\boldsymbol{z} = \boldsymbol{X}\boldsymbol{w} + \boldsymbol{b}$$

where $\boldsymbol{X}$ is a $n \times p$ matrix representing the weights, $\boldsymbol{w}$ is an $p \times q$ matrix representing the weights (q representing the number of neurons in the current layer), and $\boldsymbol{b}$ is an $n \times q$ matrix representing the biases. n represents the number of observations and p represents the dimension of the input.

Example: Calculate the Neuron Values in the First Hidden Layer

$$\boldsymbol{X} = \begin{pmatrix} 1 & 2 \\ 3 & -1 \end{pmatrix}, \boldsymbol{w} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

We can calculate the neuron value as $\boldsymbol{z}$ follows:

$$\begin{aligned} \boldsymbol{z} &= \boldsymbol{X}\boldsymbol{w} + \boldsymbol{b} \\ &= \begin{pmatrix} & \\ & \end{pmatrix} \begin{pmatrix} \\ \end{pmatrix} + \begin{pmatrix} \\ \end{pmatrix} \\ &= \begin{pmatrix} & \\ & \end{pmatrix} + \begin{pmatrix} \\ \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 8 \end{pmatrix} \end{aligned}$$

Alternatively, one can use Python:

```
import numpy as np
X = np.array([[1, 2], [3, -1]])
w = np.array([[2], [-1]])
b = np.array([[1], [1]])
print(X @ w + b)
```

```
[[1]
 [8]]
```

Exercises

1. (2×2 matrices) Calculate $\boldsymbol{z}$, given:

1. $\boldsymbol{X} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$ $\boldsymbol{w} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ $\boldsymbol{b} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$
2. $\boldsymbol{X} = \begin{pmatrix} 1 & -1 \\ 0 & 5 \end{pmatrix}$ $\boldsymbol{w} = \begin{pmatrix} -1 \\ 8 \end{pmatrix}$ $\boldsymbol{b} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$

2. (3×3 matrices) Calculate $\boldsymbol{z}$, given:
   1. $\boldsymbol{X} = \begin{pmatrix} 4 & 4 & 0 \\ 2 & 2 & 4 \\ 2 & 4 & 1 \end{pmatrix}$ $\boldsymbol{w} = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$ $\boldsymbol{b} = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}$
   2. $\boldsymbol{X} = \begin{pmatrix} 6 & -6 & -2 \\ -3 & -1 & -5 \\ 1 & 1 & -7 \end{pmatrix}$ $\boldsymbol{w} = \begin{pmatrix} 4 \\ 4 \\ -8 \end{pmatrix}$ $\boldsymbol{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$

3. (non-square matrices) Calculate $\boldsymbol{z}$, given:
   1. $\boldsymbol{X} = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 2 & 1 \end{pmatrix}$ $\boldsymbol{w} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$ $\boldsymbol{b} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$
   2. $\boldsymbol{X} = \begin{pmatrix} 1 & -1 \\ 0 & 5 \\ 2 & -2 \end{pmatrix}$ $\boldsymbol{w} = \begin{pmatrix} 5 \\ -7 \end{pmatrix}$ $\boldsymbol{b} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$

4. If $\boldsymbol{X}$ is a 2×3 matrix, what does this say about the neural network's architecture? What about a 3×2 matrix?

Activation Functions

The result of $\boldsymbol{z} = \boldsymbol{X}\boldsymbol{w} + \boldsymbol{b}$ will be in the range $(-\infty, \infty)$. However, sometimes we might want to constrain the values of $\boldsymbol{z}$. We apply an **activation function** to $\boldsymbol{z}$ to do this. Activation functions include:

- Sigmoid: $S(z_i) = \frac{1}{1+e^{-z_i}}$, constrains each value in $\boldsymbol{z}$ to $(0, 1)$
- Tanh: $\tanh(z_i) = \frac{e^{2z_i}-1}{e^{2z_i}+1}$, constrains each value in $\boldsymbol{z}$ to $(-1, 1)$.
- ReLU: $\text{ReLU}(z_i) = \max(0, z_i)$, only activates for a value of $\boldsymbol{z}$ if it is positive.
- Softmax: $\sigma(z_i) = \frac{e^{z_i}}{\Sigma_{j=1}^{K} e^{z_j}}$. This maps the values in $\boldsymbol{z}$ so that each value is in $[0, 1]$ and the sum is equal to 1. This is useful for representing probabilities and is often used for the output layer.

Example: Applying Activation Functions

Given $\boldsymbol{z} = \begin{pmatrix} 1 \\ 8 \end{pmatrix}$, calculate the resulting vector $\boldsymbol{a} = \text{activation}(\boldsymbol{z})$ using the four activation functions above.

- Sigmoid:

$$S(\boldsymbol{z}) =$$

- Tanh:

$$\tanh(\boldsymbol{z}) =$$

- ReLU

$$\text{ReLU}(\boldsymbol{z}) =$$

- Softmax

$$\sigma(\boldsymbol{z}) =$$

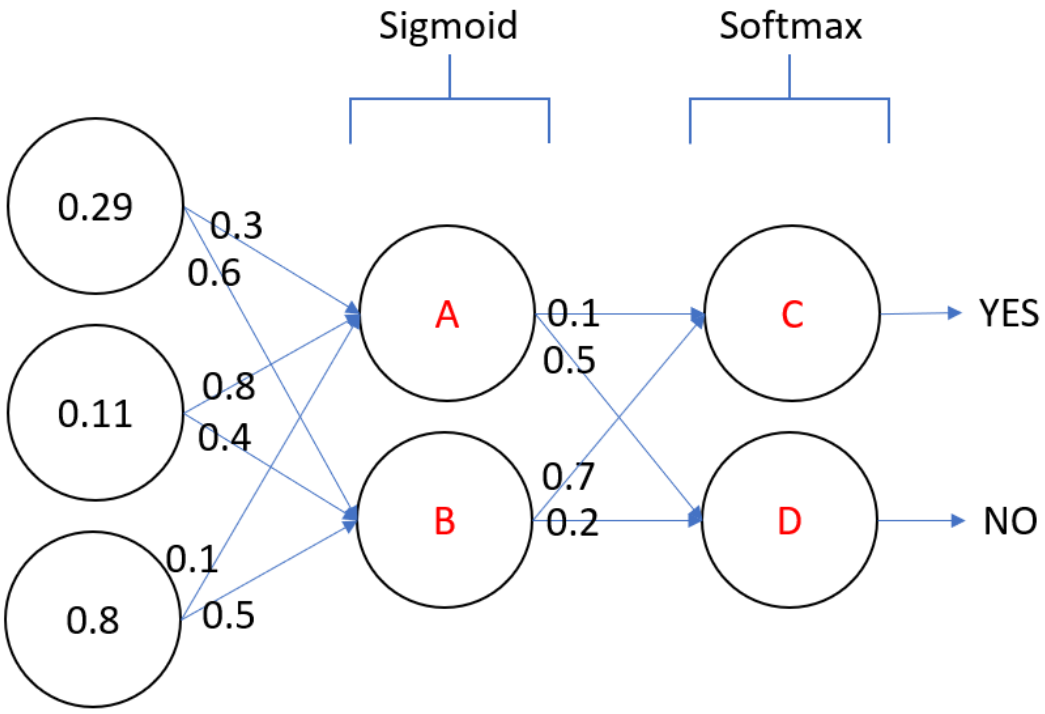
Exercises

1. Given $\boldsymbol{z} = \begin{pmatrix} 8 \\ 6 \end{pmatrix}$, calculate the resulting vector $\boldsymbol{a} = \text{activation}(\boldsymbol{z})$ using the four activation functions above.
2. Given $\boldsymbol{z} = \begin{pmatrix} -8 \\ 9 \\ -3 \end{pmatrix}$, calculate the resulting vector $\boldsymbol{a} = \text{activation}(\boldsymbol{z})$ using the four activation functions above.

3. For extra practice, try calculating the vector $\boldsymbol{a}$, using the results of the exercises in section 1.

Final Output

Example: Calculate the Final Output



1. With the activations, weights, and activation functions given in the above figure and a constant bias of 1 for each node, calculate the values of **A**, **B**, **C**, and **D**.
2. If the **C** node represents "YES" and the **D** node represents "NO", what final value is predicted by the neural network?

Hint: Write out

1. The input matrix $\boldsymbol{X}$ (should be 1×3):

$$\boldsymbol{X} = \begin{pmatrix} & & \end{pmatrix}.$$

2. The weight matrix $\boldsymbol{w}_1$ between the input layer and the first hidden layer (should be 3×2):

$$\boldsymbol{w}_1 = \begin{pmatrix} & \\ & \\ & \end{pmatrix}, \boldsymbol{b}_1 = \begin{pmatrix} & \end{pmatrix}.$$

3. The weight matrix $\boldsymbol{w}_2$ between the first hidden layer and the output layer (should be 2×2):

$$\boldsymbol{w}_2 = \begin{pmatrix} & \\ & \end{pmatrix}, \boldsymbol{b}_2 = \begin{pmatrix} & \end{pmatrix}.$$

See more details in maths-of-neural-networks.ipynb.

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

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

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:

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

The model is

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

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

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

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. ]]
[[ 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

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

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

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

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

where

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

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

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

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

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

$$\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{w}} = \boldsymbol{X}, \tag{5}$$

$$\frac{\partial \hat{\boldsymbol{y}}}{\partial b} = \mathbf{1}. \tag{6}$$

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

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

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

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

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

Subsequently,

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

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

```
#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.

$$\eta = 0.1, \tag{11}$$

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

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

```
#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))
```

```python
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)
```

```
[[1.49985359]
 [1.49973438]]
0.10057783016938827
```

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.

Backpropagation performs a backward pass to adjust the neural network's parameters. It's an algorithm that uses gradient descent to update the neural network weights.

Linear Regression via Batch Gradient Descent

Let $\boldsymbol{\theta}^{(t)} = (w^{(t)}, b^{(t)})$ be the parameter estimates of the tth iteration. Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ represents the training batch. Let mean squared error (MSE) be the loss/cost function $\mathcal{L}$.

Finding the Gradients

- **Step 1:** Write down $\mathcal{L}(\mathcal{D}, \boldsymbol{\theta}^{(t)})$ and $\hat{y}(x_i; \boldsymbol{\theta}^{(t)})$

$$\mathcal{L}(\mathcal{D}, \boldsymbol{\theta}^{(t)}) = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}) - y_i\right)^2$$
$$\hat{y}(x_i; \boldsymbol{\theta}^{(t)}) = w^{(t)} x_i + b^{(t)}$$

- **Step 2:** Derive $\frac{\partial \mathcal{L}(\hat{y}(x_i;\boldsymbol{\theta}^{(t)}), y_i)}{\partial \hat{y}(x_i;\boldsymbol{\theta}^{(t)})}$ and $\frac{\partial \hat{y}(x_i;\boldsymbol{\theta}^{(t)})}{\partial \boldsymbol{\theta}^{(t)}}$

$$\frac{\partial \mathcal{L}(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}), y_i)}{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})} = 2(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}) - y_i)$$
$$\frac{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})}{\partial w^{(t)}} = x_i$$
$$\frac{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})}{\partial b^{(t)}} = 1$$

- **Step 3:** Derive $\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta}^{(t)})}{\partial \boldsymbol{\theta}^{(t)}}$

$$\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta}^{(t)})}{\partial w^{(t)}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}), y_i)}{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})} \frac{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})}{\partial w^{(t)}} = \frac{2}{N} \sum_{i=1}^{N} \left(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}) - y_i\right) \cdot x_i \quad (14)$$

$$\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta}^{(t)})}{\partial b^{(t)}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}), y_i)}{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})} \frac{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})}{\partial b^{(t)}} = \frac{2}{N} \sum_{i=1}^{N} \left(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}) - y_i\right) \cdot 1 \quad (15)$$

Then, we initialise $\boldsymbol{\theta}^{(0)} = (w^{(0)}, b^{(0)})$ and then apply gradient descent for $t = 1, 2, \ldots$

$$w^{(t+1)} = w^{(t)} - \eta \cdot \left.\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta}^{(t)})}{\partial w}\right|_{w^{(t)}} \quad (16)$$

$$b^{(t+1)} = b^{(t)} - \eta \cdot \left.\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta}^{(t)})}{\partial b}\right|_{b^{(t)}} \quad (17)$$

using the derivatives derived from Equation 14 and Equation 15. η is a chosen learning rate.

Exercise

1. Use backpropagation algorithm to find $\theta^{(3)}$ with $\theta^{(0)} = (w^{(0)} = 1, b^{(0)} = 0)$. The dataset $\mathcal{D}$ is as follows:

i	x_i	y_i
1	2	7
2	3	10
3	5	16

That is, the true model would be $y_i = 3x_i + 1$, i.e., $w = 3, b = 1$. Implement batch gradient descent.

Neural Network

For a neural network with H hidden layers:

- L_0 is the input layer (the zeroth hidden layer). L_k represents the kth hidden layer for $k \in \{1, 2, \ldots, H\}$. L_{H+1} is the output layer (the $H+1$th hidden layer).
- $\phi^{(k)}$ represents the activation function for the kth hidden layer, with $k \in \{1, 2, \ldots, H\}$. $\phi^{(H+1)}$ represents the activation function for the output layer.
- $\boldsymbol{w}_j^{(k)}$ represents the weights connecting the activated neurons $\boldsymbol{a}^{(k-1)}$ from the $k-1$th hidden layer to the jth neuron in the kth hidden layer, where $k \in \{1, \ldots, H+1\}$ and $j \in \{1, \ldots, q_k\}$, i.e., q_k denotes the number of neurons in the kth hidden layer. $\boldsymbol{a}^{(0)} = \boldsymbol{z}^{(0)} = \boldsymbol{x}$ by definition.
- $b_j^{(k)}$ represents the bias for the jth neuron in the kth hidden layer.

Gradients For the Output Layer

The gradient for $\boldsymbol{w}_1^{(H+1)}$, i.e., the weights connecting the neurons in the Hth (last) hidden layer to the first neuron of the output layer, is given by:

$$\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial \boldsymbol{w}_1^{(H+1)}} = \frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial z_1^{(H+1)}} \frac{\partial z_1^{(H+1)}}{\partial \boldsymbol{w}_1^{(H+1)}} \tag{18}$$

where

- $\hat{y}_1 = a_1^{(H+1)} = \phi(z_1^{(H+1)})$
- $z_1^{(H+1)} = \langle \boldsymbol{a}^{(H)}, \boldsymbol{w}_1^{(H+1)} \rangle + b_1^{(H+1)}$.
- $\langle \cdot, \cdot \rangle$ represents the inner product.

Gradients For the Hidden Layers

The gradient for $\boldsymbol{w}_1^{(k)}$, i.e., the weights connecting the activated neurons $\boldsymbol{a}^{(k-1)}$ to the first neuron of the kth hidden layer $a_1^{(k)}$, is given by:

$$\begin{aligned}
\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial \boldsymbol{w}_1^{(k)}} &= \underbrace{\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial a_1^{(k)}} \frac{\partial a_1^{(k)}}{\partial z_1^{(k)}}}_{\delta_1^{(k)}} \frac{\partial z_1^{(k)}}{\partial \boldsymbol{w}_1^{(k)}} \\
&= \underbrace{\sum_{l \in \{1,\ldots,q_{k+1}\}} \frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial z_l^{(k+1)}} \frac{\partial z_l^{(k+1)}}{\partial a_1^{(k)}}}_{\text{Total Derivative}} \frac{\partial a_1^{(k)}}{\partial z_1^{(k)}} \frac{\partial z_1^{(k)}}{\partial \boldsymbol{w}_1^{(k)}} \\
&= \underbrace{\sum_{l \in \{1,\ldots,q_{k+1}\}} \delta_l^{(k+1)} w_{1,l}^{(k+1)} \frac{\partial a_1^{(k)}}{\partial z_1^{(k)}}}_{\delta_1^{(k)}} \boldsymbol{a}^{(k-1)}
\end{aligned}$$

Based on Equation 19, the derivative of the loss function with respect to the pre-activated value of the ith neuron in the kth hidden layer is given by

$$\delta_i^{(k)} = \frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial a_i^{(k)}} \frac{\partial a_i^{(k)}}{\partial z_i^{(k)}} = \sum_{l \in \{1,\ldots,q_{k+1}\}} \delta_l^{(k+1)} w_{i,l}^{(k+1)} \frac{\partial a_i^{(k)}}{\partial z_i^{(k)}}$$

Example 1

- From input layer L_0 to the first hidden layer L_1:

$$\begin{aligned}
a_1^{(1)} &= \phi^{(1)}(w_{1,1}^{(1)} x_1 + w_{2,1}^{(1)} x_2 + w_{3,1}^{(1)} x_3 + b_1^{(1)}) = \phi^{(1)}(\langle \boldsymbol{w}_1^{(1)}, \boldsymbol{x} \rangle + b_1^{(1)}) \\
a_2^{(1)} &= \phi^{(1)}(w_{1,2}^{(1)} x_1 + w_{2,2}^{(1)} x_2 + w_{3,2}^{(1)} x_3 + b_2^{(1)}) = \phi^{(1)}(\langle \boldsymbol{w}_2^{(1)}, \boldsymbol{x} \rangle + b_2^{(1)})
\end{aligned}$$

- From the first hidden layer L_1 to the output layer layer L_2:

$$\hat{y} = \phi^{(2)}(w_{1,1}^{(2)} a_1^{(1)} + w_{2,1}^{(2)} a_2^{(1)} + b_1^{(2)}) = \phi^{(2)}(\langle \boldsymbol{w}_1^{(2)}, \boldsymbol{a}^{(1)} \rangle + b_1^{(2)})$$

- $\phi^{(1)}(z) = S(z)$ (sigmoid function) and $\phi^{(2)}(z) = \exp(z)$ (exponential function).

Let $\boldsymbol{\theta}^{(t)} = (\boldsymbol{w}^{(t)}, \boldsymbol{b}^{(t)}) = \left(\boldsymbol{w}_1^{(t,1)}, \boldsymbol{w}_2^{(t,1)}, \boldsymbol{w}_1^{(t,2)}, b_1^{(t,1)}, b_2^{(t,1)}, b_1^{(t,2)}\right)$ be the parameter estimates of the tth iteration. For illustration, we assume the bias terms $(b_1^{(t,1)}, b_2^{(t,1)}, b_1^{(t,2)})$ are all zeros.

- For $\boldsymbol{w}_1^{(2)}$, apply equation Equation 18
- For $\boldsymbol{w}_1^{(1)}$, apply equation Equation 19
- For $\boldsymbol{w}_2^{(1)}$, apply equation Equation 19

Implementing Backpropagation in Python

See `Week_4_Lab_Notebook.ipynb` for more details. The required packages/functions are as follows:

```
import random
import numpy as np
import pandas as pd

from keras.models import Sequential
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.initializers import Constant
```

True weights:

```
w1_1 = np.array([[0.25], [0.5], [0.75]])
w1_2 = np.array([[0.75], [0.5], [0.25]])
w2_1 = np.array([[2.0], [3.0]])
```

Some synthetic data to work with:

```
# Generate 10000 random observations of 3 numerical features
np.random.seed(0)
X = np.random.randn(10000, 3)

# Sigmoid activation function
def sigmoid(z):
  return(1/(1+np.exp(-z)))

# Hidden Layer 1
z1_1 = X @ w1_1 # The first neuron before activation
z1_2 = X @ w1_2 # The second neuron before activation
a1_1 = sigmoid(z1_1) # The first neuron after activation
a1_2 = sigmoid(z1_2) # The second neuron after activation

# Output Layer
z2_1 = np.concatenate((a1_1, a1_2), axis = 1) @ w2_1 # Pre-activation of the ouput
a2_1 = np.exp(z2_1) # Output
```

```
# The actual values
y = a2_1
```

From Scratch

```
# Initialised weights
w1_1_hat = np.array([[0.2], [0.6], [1.0]])
w1_2_hat = np.array([[0.4], [0.8], [1.2]])
w2_1_hat = np.array([[1.0], [2.0]])

losses = []
num_iterations = 5000
for _ in range(num_iterations):
  # Compute Forward Passes
  # Hidden Layer 1
  z1_1_hat = X @ w1_1_hat  # The first neuron before activation
  z1_2_hat = X @ w1_2_hat  # The second neuron before activation
  a1_1_hat = sigmoid(z1_1_hat) # The first neuron after activation
  a1_2_hat = sigmoid(z1_2_hat) # The second neuron after activation
  a1_hat = np.concatenate((a1_1_hat, a1_2_hat), axis = 1)

  # Output Layer
  z2_1_hat = a1_hat @ w2_1_hat # The output before activation
  y_hat = np.exp(z2_1_hat).reshape(len(y), 1) # The ouput

  # Track the Losses
  loss = (y_hat - y)**2
  losses.append(np.mean(loss))

  # Compute Deltas
  delta2_1 = 2 * (y_hat - y) * np.exp(z2_1_hat)
  delta1_1 = w2_1_hat[0] * delta2_1 * sigmoid(z1_1_hat) * (1-sigmoid(z1_1_hat))
  delta1_2 = w2_1_hat[1] * delta2_1 * sigmoid(z1_2_hat) * (1-sigmoid(z1_2_hat))

  # Compute Gradients
  d2_1_hat = delta2_1 * a1_hat
  d1_1_hat = delta1_1 * X
  d1_2_hat = delta1_2 * X

  # Learning Rate
```

```
  eta = 0.0005

  # Apply Batch Gradient Descent
  w2_1_hat -= eta * np.mean(d2_1_hat, axis = 0).reshape(2, 1)
  w1_1_hat -= eta * np.mean(d1_1_hat, axis = 0).reshape(3, 1)
  w1_2_hat -= eta * np.mean(d1_2_hat, axis = 0).reshape(3, 1)

print(w1_1_hat)
print(w1_2_hat)
print(w2_1_hat)
```

```
[[0.24985576]
 [0.5000211 ]
 [0.75018656]]
[[0.74987578]
 [0.49998626]
 [0.25009692]]
[[1.99874327]
 [3.00125615]]
```

From Keras

```
# An initialiser for the weights in the neural network
init1 = Constant([[0.2, 0.4], [0.6, 0.8], [1.0, 1.2]])
init2 = Constant([[1.0], [2.0]])

# Build a neural network
# `use_bias` (whether to include bias terms for the neurons or not) is True by default
# `kernel_initializer` adjusts the initialisations of the weights
x = Input(shape=X.shape[1:], name="Inputs")
a1 = Dense(2, "sigmoid", use_bias=False,
           kernel_initializer=init1)(x)
y_hat = Dense(1, "exponential", use_bias=False,
              kernel_initializer=init2)(a1)
model = Model(x, y_hat)

# Choosing the optimiser and the loss function
model.compile(optimizer="adam", loss="mse")

# Model Training
```

```
# We don't implement early stopping to make the results comparable to the previous section
hist = model.fit(X, y, epochs=5000, verbose=0, batch_size = len(y))

# Print out the weights
print(model.get_weights())
```

```
[array([[0.25250572, 0.75124925],
       [0.49919188, 0.50062895],
       [0.74790984, 0.2485936 ]], dtype=float32), array([[2.0164018],
       [2.983464 ]], dtype=float32)]
```