GLO-7030: Deep Learning
(Apprentissage par réseaux de neurones profonds)

Practical work 1

The report is for the first practical work of the course

Prepared By

Parham NOORALISHAHI

Teacher

Dr. Pascal Germain

Université Laval
Quebec

Department of Electrical and Computer Engineering
Faculty of Science and Engineering
Université Laval

Winter Semester 2021

1 Part One : Some calculus by hand ! (40%)

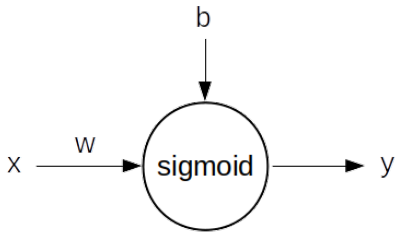
1.1 Question 1

Training Set The training set is defined as $S \in \{(x_i, y_i)\}_{i=1}^{n}$ where x_i and y_i respectively are the input and the desired output of the network, and n is the number of samples in the training set.

Loss Function The loss function is defined as:

$$F(w,b) = \frac{1}{n}\sum_{i=1}^{n} f(w,b,x_i,y_i) \text{ where}$$
$$f(w,b,x,y) = -y(w.x+b) + log(1+e^{w.x+b}) + \lambda \|w\|^2$$
$$f(w,b,x,y) = L_{nlv}\left(\sigma(w.x+b), y\right) + \lambda \|w\|^2$$

Network the network contains only one neuron with sigmoidal activation function σ and is defined as $\hat{y} = \sigma(a) = \frac{1}{1+\exp^{-a}}$ where, $a = w.x + b$



(a) Calculate $\frac{\partial L_{nlv(\hat{y},y)}}{\partial \hat{y}}$

We can calculate $\frac{\partial L_{nlv(\hat{y},y)}}{\partial \hat{y}}$ where $L_{nlv}(\hat{y},y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$. So we have,

$$\frac{\partial L_{nlv}(\hat{y},y)}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}}\left(-y\log(\hat{y}) - (1-y)\log(1-\hat{y})\right) =$$
$$\frac{-y}{\hat{y}} - \left(-\frac{1-y}{1-\hat{y}}\right) = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

Based on the above calculation, we have $\frac{\partial L_{nlv}(\hat{y},y)}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$

(b) Calculate $\frac{\partial \sigma(a)}{\partial a}$ with $\sigma(a) = \frac{1}{1+e^{-a}}$

The activation function can be written as $\hat{y} = u^{-1}$ where $u = 1 + e^{-a}$; thus based on the chain rule of derivatives, we can calculate $\frac{\partial \hat{y}}{\partial a}$ as $\frac{\partial \hat{y}}{\partial u}\frac{\partial u}{\partial a}$. So we have,

$$\frac{\partial \hat{y}}{\partial a} = \frac{\partial}{\partial a}\frac{1}{1+e^{-a}} = \frac{\partial}{\partial a}(1+e^{-a})^{-1} = -(1+e^{-a})^{-2}\frac{\partial}{\partial a}(1+e^{-a}) =$$
$$-(1+e^{-a})^{-2}(-e^{-a}) = \frac{-e^{-a}}{-(1+e^{-a})^2} = \frac{e^{-a}}{(1+e^{-a})^2} = \frac{1+e^{-a}-1}{(1+e^{-a})^2} =$$
$$\frac{1+e^{-a}}{(1+e^{-a})^2} - \frac{1}{(1+e^{-a})^2} = \frac{1}{1+e^{-a}} - \left(\frac{1}{1+e^{-a}}\right)^2 = \hat{y} - \hat{y}^2 = \hat{y}(1-\hat{y})$$

So, we have $\frac{\partial \hat{y}}{\partial a} = \hat{y}(1-\hat{y})$

(c) For $k \in \{1, ..., d\}$, calculate $\frac{\partial(w.x+b)}{\partial w_k}$ where $w = \begin{bmatrix} w_1 \\ \vdots \\ w_d \end{bmatrix}$, $x = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}$

$$\frac{\partial(w.x+b)}{\partial w_k} = x_k$$

So we have $\frac{\partial(w.x+b)}{\partial w_k} = x_k$. Therefore, for each weight, the result of $\frac{\partial(w.x+b)}{\partial w_k} = x_k$

(d) For $k \in \{1, ..., d\}$, calculate $\frac{\partial \|w\|^2}{\partial w_k}$

For calculating $\frac{\partial \|w\|^2}{\partial w_k}$ where $\|w\|^2 = w.w$, we have $\frac{\partial \|w\|^2}{\partial w_k} = 2w_k$

(e) From the previous answers, calculate $\frac{\partial f(w,b,x,y)}{\partial w_k}$

$$\frac{\partial f(w,b,x,y)}{\partial w_k} = \frac{\partial}{\partial w_k}\left(L_{nlv}(\hat{y},y) + \lambda \|w\|^2\right) =$$
$$\frac{\partial}{\partial w_k}\left(L_{nlv}(\hat{y},y)\right) + \frac{\partial}{\partial w_k}\left(\lambda \|w\|^2\right) = \frac{\partial}{\partial w_k}\left(L_{nlv}(\hat{y},y)\right) + 2\lambda w_k =$$
$$\left(\frac{\partial}{\partial \hat{y}}\left(L_{nlv}(\hat{y},y)\right)\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w_k}\right) + 2\lambda w_k = \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right).\left(\hat{y}(1-\hat{y})\right).x_k + 2\lambda w_k =$$
$$(-y(1-\hat{y}) + \hat{y}(1-y)).x_k + 2\lambda w = (\hat{y}-y).x_k + 2\lambda w_k$$

So we have $\frac{\partial f(w,b,x,y)}{\partial w_k} = (\hat{y}-y).x_k + 2\lambda w_k$

(f) From the preceding answer, deduce the expression of the gradient $\bigtriangledown_w f(w,b,x,y)$

$$\bigtriangledown_w f = \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \vdots \\ \frac{\partial f}{\partial w_d} \end{bmatrix}, \frac{\partial}{\partial w} f = \frac{\partial}{\partial w}\left(L_{nlv}(\hat{y},y) + \lambda \|w\|^2\right) =$$
$$\frac{\partial}{\partial w} L_{nlv}(\hat{y},y) + \frac{\partial}{\partial w}\lambda \|w\|^2 =$$
$$(\hat{y}-y).x + 2\lambda w$$

(g) Calculate $\frac{\partial f(w,b,x,y)}{\partial b}$, the gradient of the bias.

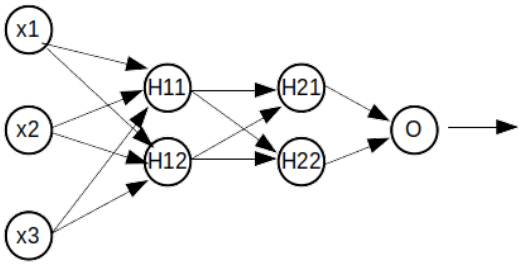
$$\frac{\partial f(w,b,x,y)}{\partial b} = \frac{\partial}{\partial b}\left(L_{nlv}(\hat{y},y) + \lambda \|w\|^2\right) = \frac{\partial}{\partial \hat{y}}\left(L_{nlv}(\hat{y},y)\right)\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial b} =$$
$\left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right).\left(\hat{y}(1-\hat{y})\right) =$ as shown previously, it can be simplified as:
$$\frac{\partial f(w,b,x,y)}{\partial b} = \hat{y} - y$$

So we have $\frac{\partial f(w,b,x,y)}{\partial b} = \hat{y} - y$.

1.2 Question 2

Network the network contains three input neurons, one output neuron, and two hidden layers that each of them contains two neurons.

$$R_\theta(x) = \sigma(b^{(3)} + W^{(3)} relu(b^{(2)} + W^{(2)} relu(b^{(1)} + W^{(1)} x)))$$



(a) forward propagation for $x = [0, 1, 2]^T$ and $y = 0$

⓪ $h^{(0)} = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$

① $a^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.1 & -1.0 & 1.0 \\ 2.0 & -0.3 & 0.1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ -0.1 \end{bmatrix}, h^{(1)} = ReLU\left(\begin{bmatrix} 1 \\ -0.1 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

② $a^{(2)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.5 & 1.0 \\ -1.0 & 2.0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ -1.0 \end{bmatrix}, h^{(2)} = ReLU\left(\begin{bmatrix} 0.5 \\ -1.0 \end{bmatrix}\right) = \begin{bmatrix} 0.5 \\ 0 \end{bmatrix}$

③ $a^{(3)} = [1] + \begin{bmatrix} -2.0 & -0.5 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} = [1.0] + [-1.0] = [0]\,,$

$$h^{(3)} = \sigma(0) = \frac{1}{1 + e^{-(0)}} = 0.5,\ \hat{y} = 0.5$$

$$J(\theta) = L_{nlv}(R_\theta(x), y) + \frac{1}{2}\|w\|^2 = -y\log(\hat{y}) - (1-y)\log(1-\hat{y}) + \frac{1}{2}\sum w_i^2 =$$
$$-\log(1 - 0.5) + 0.5 \times 16.61 = 8.998$$

(b) backpropagation

Back-propagation The extensive information regarding back-propagation is explained here.

$$w^+ = w - \eta \bigtriangledown J_\theta,\ \bigtriangledown J_\theta = \begin{bmatrix} \frac{\partial J_\theta}{\partial w_1} \\ \frac{\partial J_\theta}{\partial w_2} \\ \vdots \\ \frac{\partial J_\theta}{\partial w_k} \end{bmatrix}$$

$$J(\theta) = L_{nlv}\left(R_\theta(x), y\right) + \frac{1}{2}\|w\|^2,$$

Layer ⓪

$$g \leftarrow \bigtriangledown_{\hat{y}} L(\hat{y}, y) = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = \frac{1}{1-0.5} = 2.0$$

Layer ③

$$g \leftarrow \nabla_{a^{(3)}} J = g \odot f'(a^{(3)}) = g \odot (\hat{y}(1-\hat{y})) = 2.0 \times (0.5 \times (1-0.5)) = 0.5$$

$$\nabla_{b^{(3)}} J = g + \lambda \nabla_{b^{(3)}} \Omega(\theta) = 0.5$$

$$\nabla_{w^{(3)}} J = g h^{(2)T} + \lambda \nabla_{w^{(3)}} \Omega(\theta) = 0.5 \times \begin{bmatrix} 0.5 & 0 \end{bmatrix} + \begin{bmatrix} -2 & -0.5 \end{bmatrix} = \begin{bmatrix} -1.75 & -0.5 \end{bmatrix}$$

$$g \leftarrow \nabla_{h^{(2)}} J = w^{(3)T} g = \begin{bmatrix} -2 \\ -0.5 \end{bmatrix} \times 0.5 = \begin{bmatrix} -1.0 \\ -0.25 \end{bmatrix}$$

Layer ②

$$g \leftarrow \nabla_{a^{(2)}} J = g \odot f'(a^{(2)}) = \begin{bmatrix} -1.0 \\ -0.25 \end{bmatrix} \odot \left(\begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases} \begin{pmatrix} 0.5 \\ -1.0 \end{pmatrix} \right) = \begin{bmatrix} -1.0 \\ -0.25 \end{bmatrix} \odot \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} -1.0 \\ 0.0 \end{bmatrix}$$

$$\nabla_{b^{(2)}} J = g + \lambda \nabla_{b^{(2)}} \Omega(\theta) = \begin{bmatrix} -1.0 \\ 0.0 \end{bmatrix}$$

$$\nabla_{w^{(2)}} J = g h^{(1)T} + \lambda \nabla_{w^{(2)}} \Omega(\theta) = \begin{bmatrix} -1.0 \\ 0.0 \end{bmatrix} \times \begin{bmatrix} 1.0 & 0.0 \end{bmatrix} + \begin{bmatrix} 0.5 & 1.0 \\ -1.0 & 2.0 \end{bmatrix} = \begin{bmatrix} -1.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix} + \begin{bmatrix} 0.5 & 1.0 \\ -1.0 & 2.0 \end{bmatrix} = \begin{bmatrix} -0.5 & 1.0 \\ -1.0 & 2.0 \end{bmatrix}$$

$$g \leftarrow \nabla_{h^{(1)}} J = w^{(2)T} g = \begin{bmatrix} 0.5 & 1 \\ -1 & 2 \end{bmatrix}^T \times \begin{bmatrix} -1.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} 0.5 & -1 \\ 1 & 2 \end{bmatrix}^T \times \begin{bmatrix} -1.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} -0.5 \\ -1.0 \end{bmatrix}$$

Layer ①

$$g \leftarrow \nabla_{a^{(1)}} J = g \odot f'(a^{(1)}) = \begin{bmatrix} -0.5 \\ -1.0 \end{bmatrix} \odot \left(\begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases} \begin{pmatrix} 1 \\ -0.1 \end{pmatrix} \right) = \begin{bmatrix} -0.5 \\ -1.0 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 0 \end{bmatrix}$$

$$\bigtriangledown_{b^{(1)}} J = g + \lambda \bigtriangledown_{b^{(1)}} \Omega(\theta) = \begin{bmatrix} -0.5 \\ 0 \end{bmatrix} + 0 = \begin{bmatrix} -0.5 \\ 0 \end{bmatrix}$$

$$\bigtriangledown_{w^{(1)}} J = g h^{(0)T} + \lambda \bigtriangledown_{w^{(1)}} \Omega(\theta) =$$

$$\begin{bmatrix} -0.5 \\ 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} + \begin{bmatrix} 0.1 & -1.0 & 1.0 \\ 2.0 & -0.3 & 0.1 \end{bmatrix} =$$

$$\begin{bmatrix} 0 & -0.5 & -1.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} + \begin{bmatrix} 0.1 & -1.0 & 1.0 \\ 2.0 & -0.3 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.1 & -1.5 & 0.0 \\ 2.0 & -0.3 & 0.1 \end{bmatrix}$$

$$g \leftarrow \bigtriangledown_{h^{(0)}} J = w^{(1)T} g = \begin{bmatrix} 0.1 & -1 & 1 \\ 2 & -0.3 & 0.1 \end{bmatrix}^T \times \begin{bmatrix} -0.5 \\ 0 \end{bmatrix} =$$

$$\begin{bmatrix} 0.1 & 2 \\ -1 & -0.3 \\ 1 & 0.1 \end{bmatrix} \times \begin{bmatrix} -0.5 \\ 0 \end{bmatrix} = \begin{bmatrix} -0.05 \\ 0.5 \\ -0.5 \end{bmatrix}$$

(c) repeat (a) and (b) with different initialization

$$W^{(1)} = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}, b^{(1)} = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \end{bmatrix}, b^{(2)} = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}$$

$$W^{(3)} = \begin{bmatrix} 1.0 & 1.0 \end{bmatrix}, b^{(2)} = \begin{bmatrix} 1.0 \end{bmatrix}$$

Feedforward Apply feedforward for the new weights.

⓪ $h^{(0)} = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$

① $a^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}, h^{(1)} = ReLU\left(\begin{bmatrix} 3 \\ 3 \end{bmatrix}\right) = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$

② $a^{(2)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \end{bmatrix} \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 6.0 \\ 6.0 \end{bmatrix}, h^{(2)} = ReLU\left(\begin{bmatrix} 6.0 \\ 6.0 \end{bmatrix}\right) = \begin{bmatrix} 6.0 \\ 6.0 \end{bmatrix}$

③ $a^{(3)} = \begin{bmatrix} 1 \end{bmatrix} + \begin{bmatrix} 1.0 & 1.0 \end{bmatrix} \begin{bmatrix} 6.0 \\ 6.0 \end{bmatrix} = \begin{bmatrix} 1.0 \end{bmatrix} + \begin{bmatrix} 12.0 \end{bmatrix} = \begin{bmatrix} 13.0 \end{bmatrix},$

$$h^{(3)} = \sigma(13.0) = \frac{1}{1 + e^{-13.0}} = 0.9999, \ \hat{y} = 0.9999$$

$$J(\theta) = L_{nlv}(R_\theta(x), y) + \frac{1}{2} \|w\|^2 = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}) + \frac{1}{2} \sum w_i^2 =$$

$$-\log(1 - 0.9999) + (0.5 \times 12.0) = 15.21$$

Backpropagation Apply back-propagation for the weights.

Layer ⓪

$$g \leftarrow \bigtriangledown_{\hat{y}} L(\hat{y}, y) = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = \frac{1}{1-0.9999} = 10000$$

Layer ③

$$g \leftarrow \bigtriangledown_{a^{(3)}} J = g \odot f'(a^{(3)}) = g \odot (\hat{y}(1-\hat{y})) =$$
$$10000 \times (0.9999 \times (1-0.9999)) = 0.9999$$

$$\bigtriangledown_{b^{(3)}} J = g + \lambda \bigtriangledown_{b^{(3)}} \Omega(\theta) = 0.9999$$
$$\bigtriangledown_{w^{(3)}} J = g h^{(2)T} + \lambda \bigtriangledown_{w^{(3)}} \Omega(\theta) =$$
$$0.9999 \times \begin{bmatrix} 6.0 & 6.0 \end{bmatrix} + \begin{bmatrix} 1.0 & 1.0 \end{bmatrix} = \begin{bmatrix} 6.9994 & 6.9994 \end{bmatrix}$$
$$g \leftarrow \bigtriangledown_{h^{(2)}} J = w^{(3)T} g = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix} \times 0.9999 = \begin{bmatrix} 0.9999 \\ 0.9999 \end{bmatrix}$$

Layer ②

$$g \leftarrow \bigtriangledown_{a^{(2)}} J = g \odot f'(a^{(2)}) =$$
$$\begin{bmatrix} 0.9999 \\ 0.9999 \end{bmatrix} \odot \left(\begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases} \begin{pmatrix} 6.0 \\ 6.0 \end{pmatrix} \right) =$$
$$\begin{bmatrix} 0.9999 \\ 0.9999 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.9999 \\ 0.9999 \end{bmatrix}$$

$$\bigtriangledown_{b^{(2)}} J = g + \lambda \bigtriangledown_{b^{(2)}} \Omega(\theta) = \begin{bmatrix} 0.9999 \\ 0.9999 \end{bmatrix}$$
$$\bigtriangledown_{w^{(2)}} J = g h^{(1)T} + \lambda \bigtriangledown_{w^{(2)}} \Omega(\theta) =$$
$$\begin{bmatrix} 0.9999 \\ 0.9999 \end{bmatrix} \times \begin{bmatrix} 3.0 & 3.0 \end{bmatrix} + \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \end{bmatrix} =$$
$$\begin{bmatrix} 2.9997 & 2.9997 \\ 2.9997 & 2.9997 \end{bmatrix} + \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \end{bmatrix} = \begin{bmatrix} 3.9997 & 3.9997 \\ 3.9997 & 3.9997 \end{bmatrix}$$
$$g \leftarrow \bigtriangledown_{h^{(1)}} J = w^{(2)T} g = \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \end{bmatrix}^T \times \begin{bmatrix} 0.9999 \\ 0.9999 \end{bmatrix} = \begin{bmatrix} 1.9998 \\ 1.9998 \end{bmatrix}$$

Layer ①

$$g \leftarrow \bigtriangledown_{a^{(1)}} J = g \odot f'(a^{(1)}) =$$
$$\begin{bmatrix} 1.9998 \\ 1.9998 \end{bmatrix} \odot \left(\begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases} \begin{pmatrix} 3.0 \\ 3.0 \end{pmatrix} \right) =$$
$$\begin{bmatrix} 1.9998 \\ 1.9998 \end{bmatrix} \odot \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 1.9994 \\ 1.9994 \end{bmatrix}$$

$$\bigtriangledown_{b^{(1)}} J = g + \lambda \bigtriangledown_{b^{(1)}} \Omega(\theta) = \begin{bmatrix} 1.9994 \\ 1.9994 \end{bmatrix}$$

$$\bigtriangledown_{w^{(1)}} J = g h^{(0)T} + \lambda \bigtriangledown_{w^{(1)}} \Omega(\theta) =$$

$$\begin{bmatrix} 1.9994 \\ 1.9994 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} + \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix} = \begin{bmatrix} 1 & 2.9994 & 4.9988 \\ 1 & 2.9994 & 4.9988 \end{bmatrix}$$

$$g \leftarrow \bigtriangledown_{h^{(0)}} J = w^{(1)T} g = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}^T \times \begin{bmatrix} 1.9994 \\ 1.9994 \end{bmatrix} =$$

$$\begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \\ 1.0 & 1.0 \end{bmatrix} \times \begin{bmatrix} 1.9994 \\ 1.9994 \end{bmatrix} = \begin{bmatrix} 3.9988 \\ 3.9988 \\ 3.9988 \end{bmatrix}$$

What do you notice?

The neural network's weights are initialized here to ones. As it can be inferred from the calculations, the gradients of the network's layer almost follow the same gradient, so neurons always end up doing the same thing. This effect can cause slow learning or disturb the whole learning process and make the model incapable to learn.

1.3 Question 3

Class classification problem with "C" classes which the network has "C" neurons ($y \in \{1, ..., C\}$).

$$\hat{y} = \begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_C \end{bmatrix}, \text{ with } \hat{y}_i = softmax(a_i) = S(a_i) = \frac{e^{a_i}}{\sum_{j=1}^{C} e^{a_j}}$$

$$L_{nlv}(\hat{y}, y) = -ln(\hat{y}_y)$$

(a) Calculate $\frac{\partial}{\partial a_y} L_{nlv}(\hat{y}, y)$ for the output with the right class $(y = i)$

$$\frac{\partial}{\partial a_y} softmax(a_y) = \frac{\partial}{\partial a_y} \frac{e^{a_y}}{\sum_{j=1}^{C} e^{a_j}} =$$

$$\frac{e^{a_y} \sum_{j=1}^{C} e^{a_j} - (e^{a_y})^2}{(\sum_{j=1}^{C} e^{a_j})^2} = \left(\frac{e^{a_y}}{\sum_{j=1}^{C} e^{a_j}} \right) - \left(\frac{e^{a_y}}{\sum_{j=1}^{C} e^{a_j}} \right)^2 =$$

$$S(a_y) - (S(a_y))^2 = S(a_y)\,(1 - S(a_y))$$

So we will have,

$$\frac{\partial}{\partial a_y} L_{nlv}(\hat{y}, y) = \frac{\partial}{\partial a_y} \ln \left[\frac{1}{S(a_y)} \right] =$$

$$\frac{\partial}{\partial a_y} \ln \left[S(a_y)^{-1} \right] = -\frac{\partial}{\partial a_y} \ln \left[S(a_y) \right] = \frac{-1}{S(a_y)} \times \frac{\partial}{\partial a_y} S(a_y) =$$

$$\frac{-1}{S(a_y)} \left(S(a_y)\,(1 - S(a_y)) \right) = S(a_y) - 1$$

The result is $\frac{\partial}{\partial a_y} L_{nlv}(\hat{y}, y) = softmax(a_y) - 1$.

(b) Calculate $\frac{\partial}{\partial a_i} L_{nlv}(\hat{y}, y)$ for the output with the right class $(y \neq i)$

$$\frac{\partial}{\partial a_i} softmax(a_y) = \frac{\partial}{\partial a_i} \frac{e^{a_y}}{\sum_{j=1}^{C} e^{a_j}} =$$

$$\frac{0 - e^{a_i} e^{a_y}}{\left(\sum_{j=1}^{C} e^{a_j}\right)^2} = \left(\frac{-e^{a_i}}{\sum_{j=1}^{C} e^{a_j}}\right)\left(\frac{e^{a_y}}{\sum_{j=1}^{C} e^{a_j}}\right) = -S(a_i)S(a_y)$$

So we will have,

$$\frac{\partial}{\partial a_y} L_{nlv}(\hat{y}, y) = \frac{\partial}{\partial a_y} \ln\left[\frac{1}{S(a_y)}\right] =$$

$$\frac{\partial}{\partial a_y} \ln\left[S(a_y)^{-1}\right] = -\frac{\partial}{\partial a_y} \ln\left[S(a_y)\right] = \frac{-1}{S(a_y)} \times \frac{\partial}{\partial a_y} S(a_y) =$$

$$\frac{-1}{S(a_y)} \left(-S(a_i)S(a_y)\right) = S(a_i)$$

The result is $\frac{\partial}{\partial a_i} L_{nlv}(\hat{y}, y) = softmax(a_y)$.

(c) From the (a) and (b), find the gradient $\bigtriangledown_a L_{nlv}(\hat{y}_y, y)$

$$\bigtriangledown_a L(\hat{y}, y) = \begin{bmatrix} \frac{\partial L}{\partial a_1} \\ \frac{\partial L}{\partial a_2} \\ \vdots \\ \frac{\partial L}{\partial a_C} \end{bmatrix}, \frac{\partial}{\partial a} L(\hat{y}, y) = \frac{\partial}{\partial a} \ln\left[\frac{1}{softmax(a)}\right] = \begin{cases} S(a_i) - 1 & \text{, where } i = y \\ S(a_i) & \text{, where } i \neq y \end{cases} = softmax(a) - y$$

So we have $softmax(a) - y$.

2 Part Two : Experiments with PyTorch (60%)

This part contains the required description for second part of TP1. The implementations and results are explained here in details. Moreover, for each question, a folder is added to the project path containing the trained model, the Poutyne experiment, and the results.

Files	Description
`parham_core.py`	This file contains all the functions and classes required for the questions 4 and 5.
`question4_{exp1|exp2}.py`	The files contain the implementation of question 4 for experiment 1 and 2.
`question5a.py`	This file contains the implementation of question 5a.
`question5b.py`	This file contains the implementation of question 5b.

3 Question 4

In this question, a neural network containing fully-connected layers is used to learn the MNIST dataset for handwritten number recognition. Since the input data is a 28×28 matrix but the network involves linear fully-connected layers. This data needs to be transformed into an array with 784 items. To do so, a flatten layer is added to the network transforming the data into an array with the mentioned size. Moreover, four fully-connected layers are added to the network respectively with 256, 128, 64, 10 neurons. Since the network supposes to recognize handwritten numbers, the output will be a number between zero to nine. The chosen loss function is a combination of log-softmax and cross-entropy and the optimizer is SGD. The network architecture is shown in Figure 1 and details of the layers are presented in Table 1.

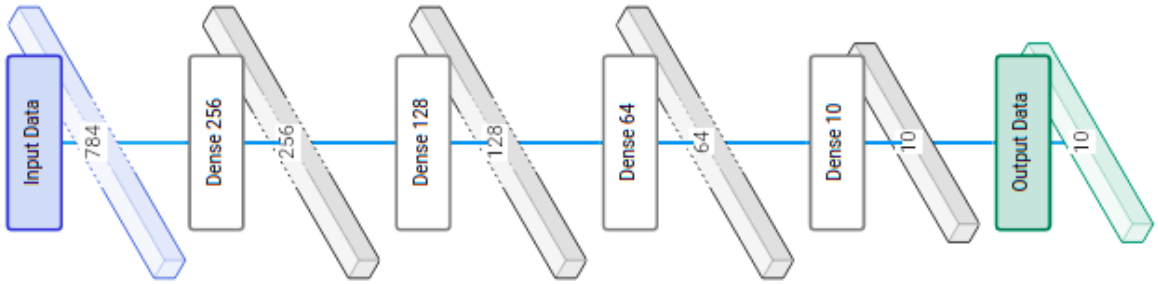


Figure 1: A diagram showing the overall architecture of the designed model.

For this section, two experiments are conducted with same parameters with different subset of MNIST dataset to analyze the results. In the first experiment, the whole training set is used to train the model and the test dataset is used as a validation dataset. For the second experiment, 0.8 portion of training dataset is set as training subset and 0.2 as validation subset. The details about model's parameters and experiments' configurations are listed in Table 2. Moreover, Table 3 presents files and figures related to the experiments.

Table 1: The details about the model's layers and the employed activation function.

L#	Type	Dimension	Activation
L1	Flatten	out : 784	–
L2	Linear	784×256	ReLU
L3	Linear	256×128	ReLU
L4	Linear	128×64	ReLU
L5	Linear	64×10	ReLU
Loss Function	The combination of Log Softmax and Cross Entropy Loss		
Optimizer	SGD		

Table 2: Parameters used for training and the information regarding the dataset and how the subsets were determined. * EXP1: dataset parameters for the experiment 1, EXP2: dataset parameters for the experiment 2.

	Values
Batch Size	32
Learning Rate	0.2
Number of Epoch	60
Number of Classes	10

	EX1*	**EX2***
Number of Data	70000	60000
Number of Training Data	60000	48000
Number of Testing Data	10000	12000
Dataset Split Rate	0.0	0.8

Table 3: The list of resulted files.

File	**Description**
`final_result/question_4/q4_result_{exp1|exp2}.gif`	Animated figures demonstrating the progress of training process over different epochs.
`final_result/question_4/q4_result_{exp1|exp2}.png`	Figures demonstrating the loss and accuracy of the model during the training.
`final_result/question_4/training_result_{exp1|exp2}.tsv`	TSV files containing the information regarding the training process.
`final_result/question_4/testing_result_{exp1|exp2}.tsv`	TSV files containing the information regarding the testing process.
`results/model_q4_{exp1|exp2}.phm`	The saved trained models.

3.1 Experiment 1 : 100% of the training dataset

In the first experiment, the whole training dataset is employed to train the model and testing dataset is used for validation purposes. As shown in Figure 2, the model reached 99.9983% accuracy rate in the 33rd epoch and stayed steady over the remaining epochs. After training, the resulted loss and accuracy for testing respectively 0.09546 and 98.53.

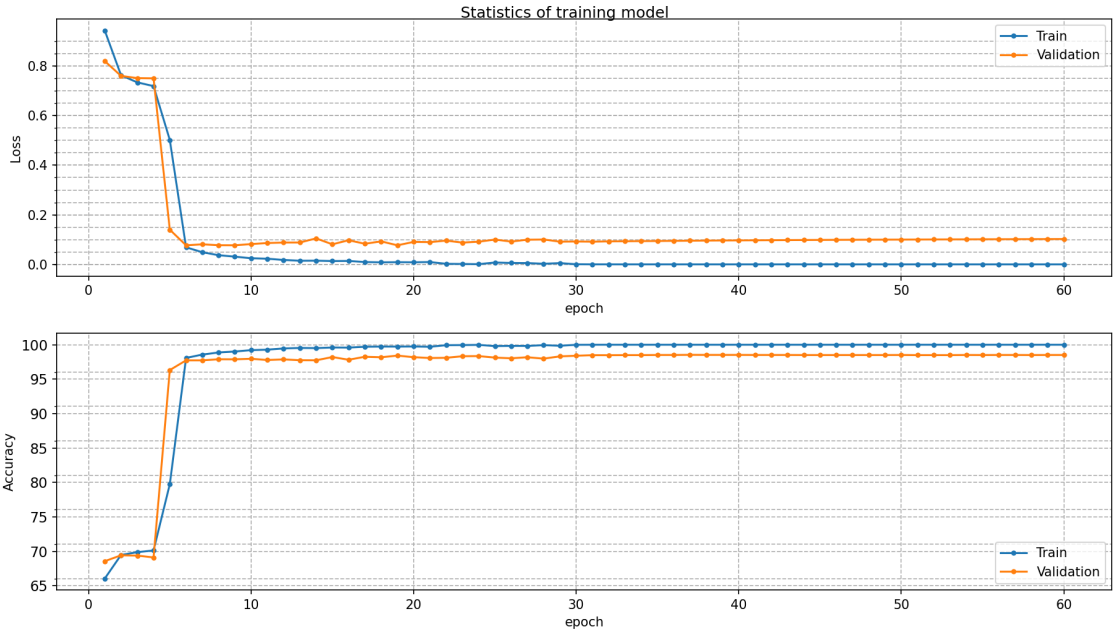


Figure 2: The figure demonstrate the loss and accuracy of the model during the training for each epoch in Experiment 1.

As mentioned in the figure, the model reached 99.9983% accuracy rate in the training, which based on the discussion in the course's forum would be a sufficient result for this question. The analysis of training and validation progress demonstrates that after the 33rd epoch, the network keeps almost steady values for training and testing accuracy and validation. This occurrence shows no sign of *overfitting* in the network since the accuracy keeps steady value and the loss doesn't decrease significantly.

3.2 Experiment 2 : 80% of the training dataset

In the second experiment, the training dataset is split with 0.8 rate to train the model and testing dataset is used for validating the dataset. As shown in Figure 3, the network reached the 100% accuracy rate during the training after *51 epoch*. After training, the resulted loss and accuracy for testing respectively are 0.1121 and 98.52.

The figure also shows that after the model reaches the 100% in training, the validation accuracy keeps a steady value while the validation loss is increasing and training accuracy is outperforming. This occurrence can be explained as *overfitting*! Overfitting happens when the model starts memorizing the training samples and picking up on noises and extract the features based on them also. It improves the accuracy of the training process but causes lower testing accuracy and an increase in validation and testing loss.

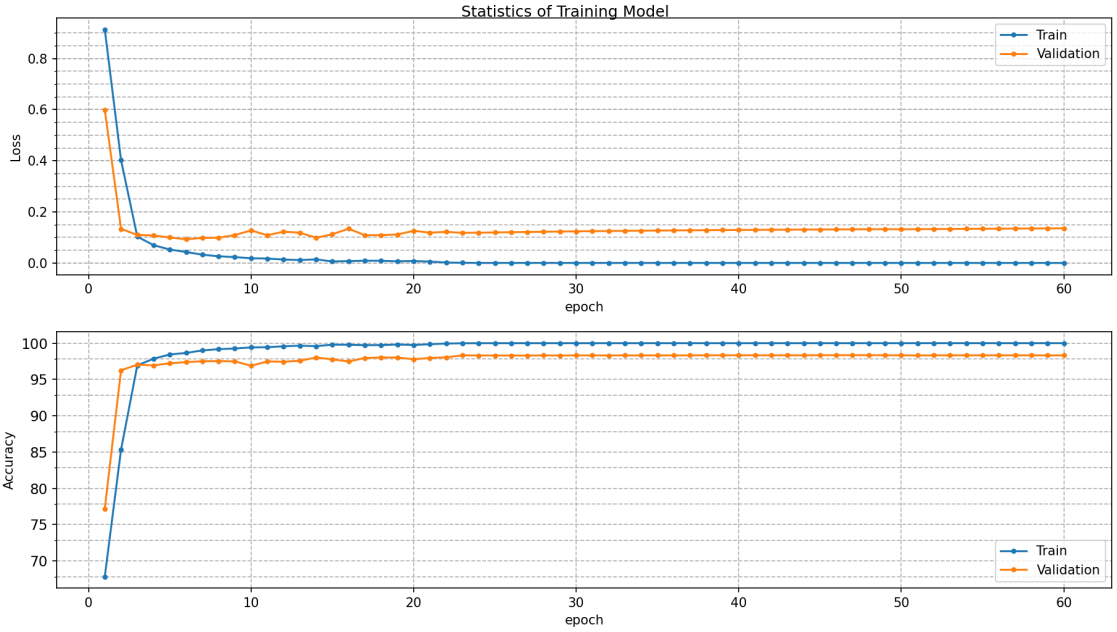


Figure 3: The figure demonstrate the loss and accuracy of the model during the training for each epoch in Experiment 2.

4 Question 5

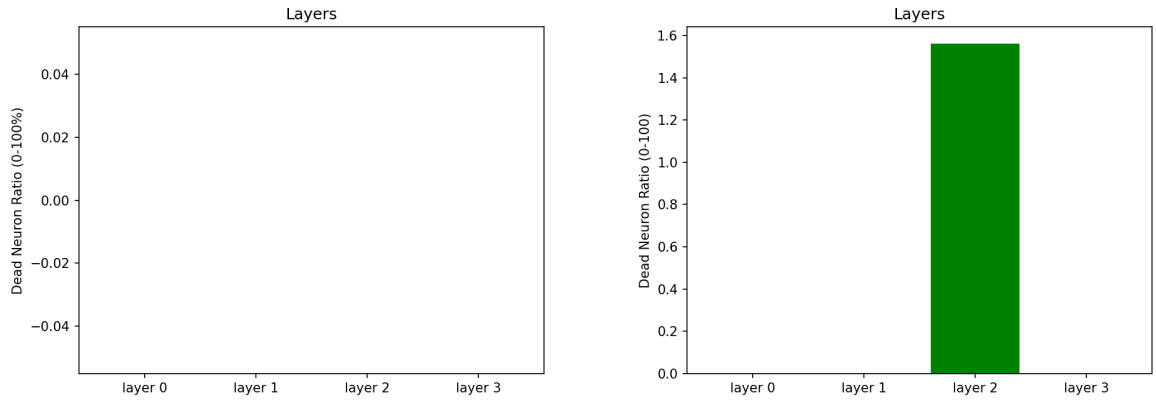
4.1 Question 5a

In this section, the results related to Question 5a are presented. The implementations for this part are located in `question5a.py` and the reusable codes are placed in `parham_core.py`. The main objective for this section is to measure the ratio of dead neurons for each layer in a model containing ReLU layers. Table 4 shows the implemented methods and classes associated to the objectives defined in this question.

Table 4: The list of methods used for the calculation of dead neuron's ratios.

Method/Class	**Description**
`DeadNeuronHook`	The class handlers for the defined hooks.
`DeadNeuronHook/calculate_dead_rate`	Calculate the required statistics regarding the ratio of dead neurons in the registered layer.
`DeadNeuronHook/hook_backward_dead_neuron`	The implemented hook for backward pass, this method is invoked by pytorch in each iteration for the registered layer. It provides access to the gradients of the layer's neurons.
`DeadNeuronHook/hook_forward_dead_neuron`	The implemented hook for forward pass, this method is invoked by pytorch in each iteration for the registered layer. It provides access to the outputs of the layer's neurons (before applying activation function).
`calculate_dead_neoron_ratio`	This method provides the required statistics.

To implement this method, the built-in hooks available in the Pytorch library are employed. Hooks are the handlers associated with the model's layers and have two types for forward and backward passes. These handlers are called when a layer is called to calculate the output based on the given input during forward



(a) Experiment 1*

(b) Experiment 2

Figure 4: The figure shows a sample result of the implemented method to detect dead neurons for the experiment 1 and 2 in Question 4. * As it is shown, the trained model in the first experiment, doesn't have any dead neuron.

pass or to calculate gradients during the backward pass. For this section, both of the forward and backward pass hooks are implemented. The backward pass hook is chosen as the primary method to calculate the ratio of dead neurons for all layers, and the forward pass is employed to test the result. Based on the conducted experiment, both approaches returns identical results. In the following, a sample result of this method is shown in the Figure 4.

Dead Neurons Dead neurons can be defined as the neurons that have never been activated for all samples in the dataset.

4.2 Question 5b

In this section, the results for Question 5b are presented. In this part, the defined model in Question 4 is used with the following initialization strategies for weight initialization: (a) Uniform(-1,1), Normal(0,1), Constant(0.1), Xavier_Normal(1), and Kaiming_Uniform(1). In Table 5, the initialization methods used for each experiment is listed.

Table 5: The initialization methods used for weights and biases in each experiment.

Ex.	**Weight**	**Bias**
#1	Uniform(-1, 1)	Uniform(-1, 1)
#2	Normal(0, 1)	Normal(0, 1)
#3	Constant(0.1)	Constant(0.1)
#4	Xavier_Normal(1)	Constant(0.0)*
#5	Kaiming_Uniform(1)	Constant(0.0)*

* The initialization method for bias is different than weight, due to the limitation on using Xavier and Kaiming methods when the number of items are only two.

The percentage of dead neurons is measured before and after the training process for each initialization strategy. Moreover, same as Question 4, the training, validation, and testing results are provided for each initialization strategy. Moreover, the training is performed in 55 epochs while the dataset is split with 0.8 ratios. The batch size and learning rate are set respectively 32 and 0.2. Moreover, for measuring the ratio

of dead neurons, a testing subset of the dataset containing random 256 samples is created to be used during the process. Beside the results presented in the report, all related files and results are placed in a folder explained in Table 6.

Table 6: The list of files containing the results.

File	**Description**
`final_result/question_5b/results/*`	All details and results in JSON format.
`final_result/question_4/saved_experiments/*`	All the conducted experiments.
`final_result/question_4/trained_models/*`	The files containing the trained models ready to load!

Which initialization method appears to be the best? In Figure 5, the results of training and validating the model developed in Question 4 are presented for all the initialization strategies. As demonstrated in Table 7, *Xavier_Normal* and *Kaiming_Uniform* method provide the most significant results compared to the other methods. **Kaiming_Uniform** presents higher accuracy in training, validation, and testing phases while shows a similar loss value compare to *Xavier_Normal*.

Table 7: A comparative analysis of different weight initialization strategies.

	E*	**DNB***				**DNA***				**Test**		**Training**		**Valid.**	
		L1	**L2**	**L3**	**L4**	**L1**	**L2**	**L3**	**L4**	**Loss**	**Acc.**	**Loss**	**Acc.**	**Loss**	**Acc.**
Xavier_Normal(1)	24	0.0	**0.78**	1.56	0.0	**0.39**	0.0	0.0	0.0	0.077	98.82	**0.00005**	**99.997**	**0.1**	98.21
Normal(0, 1)	–	0.0	0.0	0.0	0.0	100	100	100	100	2.3	7.42	2.3	9.89	2.3	9.76
Constant(0.1)	–	0.0	0.0	0.0	0.0	100	100	100	0.0	2.29	13.67	2.3	10.99	2.3	9.7
Uniform(-1, 1)	–	0.0	1.56	1.56	0.0	100	100	100	100	2.3	7.42	2.3	9.89	2.3	9.76
Kaiming_Uniform(1)	21	0.0	**0.781**	0.0	0.0	0.781	0.0	0.0	0.0	**0.042**	**99.21**	**0.00006**	**99.997**	**0.11**	**98.24**

E: Epoch id that the model reaches the highest accuracy in the training
DNB: The ratio of dead neurons before training.
DNA: The ratio of dead neurons after training.

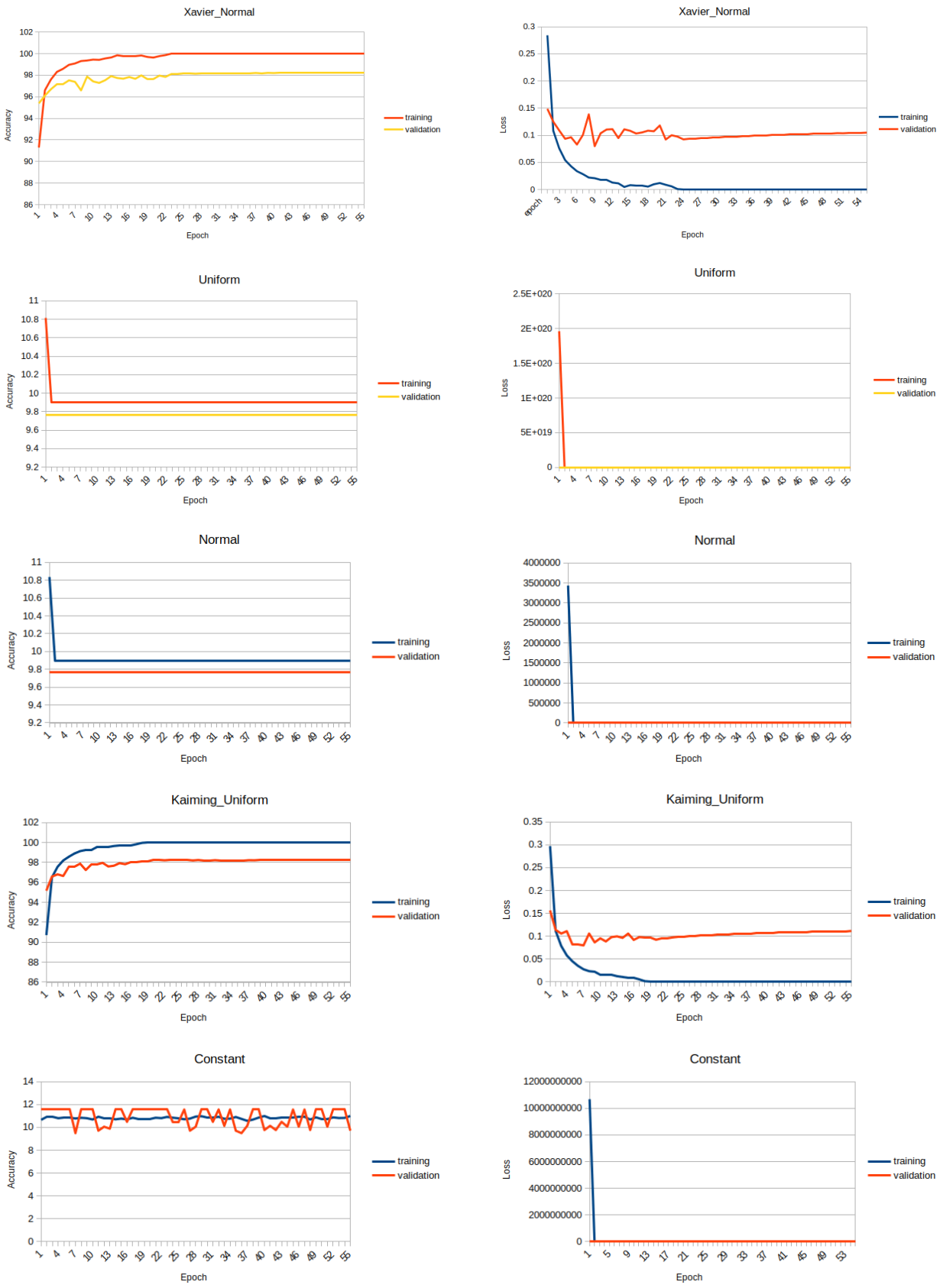


Figure 5: The loss and accuracy of the model during the training process for all the targeted weight initialization strategies.