Aprendizaje Profundo (2024-2025)
Máster en Robótica e Inteligencia Artificial
Universidad de León

Práctica 2

Sheila Martínez Gómez
Alejandro Mayorga Caro
Ángel Morales Romero

13 de noviembre de 2024

1. Cuestión 1

1.1. Enunciado

1.2. Solución

$$w_0 = f^{-1}(x)$$
$$w_0(x \cdot e^x) = x$$

$$w_0 = 0 \Rightarrow w_0(0 \cdot e^0) = 0$$
$$x = 0,0000$$
$$y = 0,0000$$

$$w_0 = 1 \Rightarrow w_0(1 \cdot e^1) = 1$$
$$x = 2,7183$$
$$y = 1,0000$$

$$w_0 = 2 \Rightarrow w_0(2 \cdot e^2) = 1$$
$$x = 14,7181$$
$$y = 2,0000$$

$$X = \begin{bmatrix} 0,0000 \\ 2,7183 \\ 14,7781 \end{bmatrix} \; Y = \begin{bmatrix} 0,0000 \\ 1,0000 \\ 2,0000 \end{bmatrix}$$

2. Cuestión 2

2.1. Enunciado

1. Entrenar la red neuronal de la figura 2 para la predicción de W_0 de acuerdo a los siguientes requisitos:
   - La red dispone de una primera capa lineal con 3 neuronas, cuyos pesos iniciales son $\begin{pmatrix}0{,}15 & -0{,}10 & 0{,}12\end{pmatrix}$ y sus sesgos $\begin{pmatrix}0{,}3 & -0{,}20 & 0{,}07\end{pmatrix}$.
   - La siguiente capa consta de funciones de activación sigmoidal.
   - La salida de la red procede de una siguiente capa lineal con 1 neurona, con pesos iniciales $\begin{pmatrix}1{,}4\\7{,}8\\3{,}4\end{pmatrix}$ y sesgo 0,5.
2. El entrenamiento se realiza utilizando como función de coste

$$C = \frac{1}{m}\sum_{i=0}^{m}(W_0(x_i) - \hat{y}_i)^2,$$

   siendo $\hat{y}_i$ la predicción de la red para el dato de entrada x_i, y m el número de datos de entrenamiento.
3. El entrenamiento se realiza utilizando la estrategia de descenso por gradiente con momento, utilizando para la primera iteración momento nulo al no disponer de mejor valor.
4. Ajustar los parámetros de la red 1 época utilizando la técnica de diferenciación automática en modo reverse con tasa de aprendizaje igual a 0,01.

2.2. Solución

Para resolver la función de Lambert mediante redes neuronales, se implementó una red neuronal simple en Python utilizando el algoritmo de backpropagation con ajuste de pesos mediante descenso de gradiente. Este modelo se entrena con un conjunto de datos de entrada X y salidas esperadas Y, optimizando los pesos y sesgos a través de 1 época.

2.3. Estructura de la Red Neuronal

La red neuronal consiste en tres capas principales:

- Una capa de entrada, donde se toman las entradas X y se multiplican por los pesos iniciales W_u y sesgos b_u.
- Una capa oculta, que aplica la función sigmoide, definida como:

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

  para introducir no linealidades en el modelo.
- Una capa de salida, que aplica los pesos finales W_y y el sesgo b_y para generar la predicción de salida.

2.4. Entrenamiento del Modelo

El entrenamiento se realiza mediante un bucle de *forward* y *backward pass* a lo largo de un número definido de épocas. Durante el *forward pass*, se calculan las predicciones y se computa el costo de la red usando la función de error cuadrático medio:

$$\text{cost} = \frac{1}{m} \sum_{i=1}^{m} (Y - \hat{Y})^2$$

donde m es el número de ejemplos de entrenamiento.
En el *backward pass*, se derivan los gradientes respecto a los pesos y sesgos usando la regla de la cadena y se actualizan mediante descenso de gradiente con momento, con una tasa de aprendizaje lr. La actualización de pesos y sesgos se realiza de la siguiente forma:
1. **Cálculo del término de momentum**: Este término suaviza la actualización de los pesos, utilizando la actualización anterior y la nueva derivada de la función de costo. La fórmula para el término de momentum es:

$$W_m := \beta \cdot W_m + (1 - \beta) \cdot \frac{\partial \text{cost}}{\partial W}$$

donde los términos de momento ayudan a suavizar las actualizaciones para evitar oscilaciones.
Donde:

- W_m es el término de momentum (que representa la actualización suavizada de los pesos).
- β es el factor de momentum, que controla cuánto de la actualización anterior se conserva. El valor de β es típicamente cercano a 1 (por ejemplo, $\beta = 0{,}9$) aunque para nuestro problema al ser más simple hemos optado por $\beta = 0{,}5$.
- $\frac{\partial \text{cost}}{\partial W}$ es el gradiente de la función de costo con respecto al peso W.

2. **Actualización de los pesos**: Finalmente, los pesos se actualizan utilizando el término de momentum y la tasa de aprendizaje lr:

$$W := W - lr \cdot W_m$$

Donde:

- W es el peso que se está actualizando (por ejemplo, Wu, bu, Wy, by).
- lr es la tasa de aprendizaje, un parámetro que controla el tamaño del paso en la actualización ($lr = 0{,}01$ en nuestro caso).
- W_m es el término de momentum calculado en el primer paso.

Este proceso se repite para cada iteración del entrenamiento, actualizando progresivamente los pesos de la red neuronal para minimizar la función de costo.

2.5. Uso del Modelo Entrenado

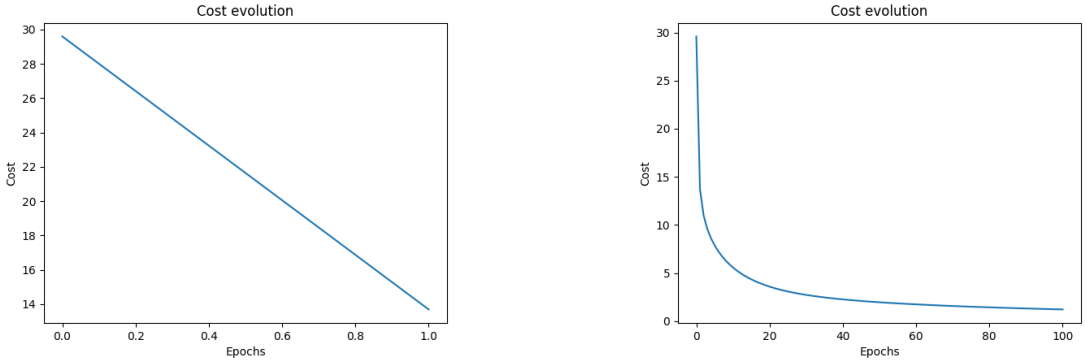
Una vez finalizado el entrenamiento, se guardan los pesos y sesgos entrenados en un archivo `pickle` para su reutilización en predicciones posteriores. La función `lambert` carga estos pesos entrenados y realiza el cálculo para cualquier nueva entrada X siguiendo la misma estructura de capas.
El código en Python implementa el entrenamiento y guardado del modelo en `train_lambert`, mientras que la predicción se realiza en la función `lambert`.

2.6. Resultados

El modelo fue entrenado con $X = \begin{bmatrix} 0{\cdot}e^0 \\ 1{\cdot}e^1 \\ 2{\cdot}e^2 \end{bmatrix}$ y $Y = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$ durante un una época.
El modelo fue entrenado por 1 época, utilizando una tasa de aprendizaje de 0.01 y un momento (momentum) para mejorar la convergencia. Los resultados del modelo entrenado son los siguientes:

- **Coste final:** 13.69094522
- **Pesos y sesgos aprendidos:**
  - $Wu = \begin{bmatrix} 0{,}08159339 & -0{,}68622434 & -0{,}11742454 \end{bmatrix}$
  - $bu = \begin{bmatrix} 0{,}27116171 & -0{,}37917983 & -0{,}00759286 \end{bmatrix}$
  - $Wy = \begin{bmatrix} 1{,}32598667 \\ 7{,}76211242 \\ 3{,}33274683 \end{bmatrix}$
  - $by = [0{,}39333115]$

La evolución del costo en cada época se graficó para observar la convergencia del modelo. A continuación, se muestra la reducción de costo, para 1 época (como se especificaba en el problema) y para 100 épocas para visualizar mejor la reducción del coste.



(a) 1 Época (b) 100 Épocas

Figura 2.1: Comparativa con diferente cantidad de épocas

3. Cuestión 3

3.1. Enunciado

Utilizando la red anterior para el cálculo de los valores objetivo de V_D en tres puntos ($V_{cc} = 3, V_{cc} = 6, V_{cc} = 9$), entrenar la red neuronal de la figura 3 para la predicción de V_D en función de V_{cc}, con requisitos:

- La red dispone de una primera capa lineal con 3 neuronas, cuyos pesos iniciales son

  $$\begin{pmatrix} 0{,}05 & 0{,}15 & -0{,}20 \end{pmatrix}$$

  y sus sesgos

  $$\begin{pmatrix} 0{,}23 & -0{,}10 & 0{,}17 \end{pmatrix}.$$

- La siguiente capa consta de funciones de activación sigmoidal.

- La capa siguiente es lineal, con 2 neuronas, cuyos pesos iniciales son

  $$\begin{pmatrix} 0{,}8 & -0{,}6 \\ 0{,}7 & 0{,}9 \\ 0{,}5 & -0{,}6 \end{pmatrix}$$

  y sus sesgos

  $$\begin{pmatrix} 0{,}45 & -0{,}34 \end{pmatrix}.$$

- La siguiente capa consta de funciones de activación ReLU.

- La salida de la red procede de una siguiente capa lineal con 1 neurona, con pesos iniciales

  $$\begin{pmatrix} 0{,}8 \\ 0{,}5 \end{pmatrix}$$

  y sesgo 0,7.

- El entrenamiento se realiza utilizando como función de coste

  $$C = \frac{1}{m} \sum_{i=0}^{2} (V_{D_i} - \hat{y}_i)^2,$$

  siendo $\hat{y}_i$ la predicción de la red para el dato de entrada V_{cc_i}, y m el número de datos de entrenamiento.

- El entrenamiento se realiza utilizando la estrategia de descenso por gradiente con momento, utilizando para la primera iteración momento nulo al no disponer de mejor valor.

- Ajustar los parámetros de la red 1 época utilizando la técnica de diferenciación automática en modo reverse.

- Valores típicos, para un diodo de germanio, podrían ser:
  - $I_0 = 10^{-12}\,\text{A}$,
  - $\eta = 1$,
  - $V_T = 0{,}026\,\text{V}$.

- Considere como valor de la resistencia $R = 100\,\Omega$.

3.2. Solución

En este ejercicio, hemos utilizado dos aproximaciones similares, tan solo cambiando la función de Lambert, pero manteniendo el esquema principal a la hora de calcular los gradientes y actualizar los pesos:

- **Aproximación 1:** Usamos la función `lambertw` de la librería `scipy.special`, que proporciona una solución directa al problema de Lambert, basada en un enfoque numérico.
- **Aproximación 2:** Implementamos una solución utilizando redes neuronales. En este caso, creamos una función `lambert` basada en una red neuronal entrenada previamente, que fue definida en el archivo `P2E2.py`. Esta aproximación es una aproximación de aprendizaje automático para modelar el problema. Debido al pequeño número de muestras y al reducido entrenamiento (1 epoch), entendemos que no será tan precisa como la función `lambertw` de la librería `scipy.special`.

Para resolver el ejercicio empezamos por calcular los valores objetivo de V_D en los puntos ($V_{CC} = 3$, $V_{CC} = 6$, $V_{CC} = 9$).
Para ello partiremos de la siguiente ecuación deducida en el enunciado:

$$V_D = V_{cc} + RI_0 - \eta V_T W_0 \left(\frac{RI_0}{\eta V_T} e^{\frac{V_{cc}+RI_0}{\eta V_T}} \right),$$

3.3. Resolución Analítica del Voltaje V_D

Definimos los parámetros:

$$a = V_{cc} + RI_0, \quad b = \frac{RI_0}{\eta V_T}, \quad \text{y} \quad c = \frac{1}{\eta V_T}.$$

La solución para V_D en función de V_{cc} es:

$$V_D = a - \eta V_T W_0 \left(b e^{ac} \right).$$

Para obtener las soluciones utilizaremos Python, declarando las constantes definidas por el enunciado y calculando el resultado del voltaje V_D en cada punto de V_{CC}.

```
# Constantes
I0 = 1e-12
eta = 1
VT = 0.026
R = 100
Vcc_values = [3, 6, 9]

# Calculo de VD
def calculate_VD_steps(Vcc):
    a = Vcc + R * I0  # Calculamos 'a'
    b = R * I0 / (eta * VT)  # Calculamos 'b'
    exponent = (Vcc + R * I0) / (eta * VT)  # Calculamos el exponent
    W_argument = b * np.exp(exponent)  # Calculamos el argumento de la funcion
```

```
    W_result = lambertw(W_argument).real  # Lambert de scipy.special
    VD = a - eta * VT * W_result  # Calculo final de VD
    return VD

# Calculamos el VD de cada Vcc
VD = {Vcc: calculate_VD_steps(Vcc) for Vcc in Vcc_values}
```

Obteniendo de este modo los siguientes valores:

$$\begin{pmatrix} 3 & 6 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 0{,}6212 & 0{,}6423 & 0{,}6538 \end{pmatrix}$$

3.4. Modelado con Red Neuronal

Para aproximar el valor de V_D utilizando una red neuronal, empleamos una arquitectura de tres capas, como se puede ver en la figura 3 del enunciado. Comenzamos por declarar las entradas $X = \begin{bmatrix} V_{cc} \end{bmatrix}$ y los pesos y sesgos para cada capa como se detalla a continuación.

3.4.1. Primera Capa

La salida de la primera capa es:

$$S = W_1 \times X + b_1,$$

donde W_1 y b_1 son los pesos y el sesgo de la primera capa, definidos como:

$$W_1 = \begin{bmatrix} 0{,}05 \\ 0{,}15 \\ -0{,}20 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0{,}23 \\ -0{,}10 \\ 0{,}17 \end{bmatrix}.$$

Aplicamos la función de activación sigmoide a S, obteniendo:

$$T = \sigma(S) = \frac{1}{1 + e^{-S}}.$$

3.4.2. Segunda Capa

La segunda capa toma como entrada T y produce una salida U mediante la operación:

$$U = W_2 \times T + b_2,$$

donde W_2 y b_2 son los pesos y el sesgo de la segunda capa, dados por:

$$W_2 = \begin{bmatrix} 0{,}8 & -0{,}6 & 0{,}5 \\ 0{,}7 & 0{,}9 & -0{,}6 \end{bmatrix}, \quad b_2 = \begin{bmatrix} 0{,}45 \\ -0{,}34 \end{bmatrix}.$$

Aplicamos la función de activación ReLU a U para obtener V:

$$V = \text{ReLU}(U) = \text{máx}(0, U).$$

3.4.3. Capa de Salida

Finalmente, la capa de salida produce el valor aproximado de V_D como:

$$Y_{pred} = W_3 \times V + b_3,$$

donde $W_3 = \begin{bmatrix}0{,}8 & 0{,}8\end{bmatrix}$ y $b_3 = 0{,}7$. En esta capa, no aplicamos una función de activación adicional, obteniendo la salida directamente como Y_{pred}.
Declarando esto que hemos comentado en Python queda de la siguiente manera:

```
# Datos y objetivos
X = np.array([[3], [6], [9]])   # Entradas Vcc (3 filas x 1 columna)
Y = np.array([[VD[3]], [VD[6]], [VD[9]]])  # Objetivo VD
alpha = 0.01  # Tasa de aprendizaje

# Pesos y sesgos iniciales
# Primera capa
W1 = np.array([[0.05], [0.15], [-0.20]])  # (3x1)
b1 = np.array([[0.23], [-0.10], [0.17]])  # (3x1)

# Segunda capa
W2 = np.array([[0.8, -0.6, 0.5], [0.7, 0.9, -0.6]])  # (2x3)
b2 = np.array([[0.45], [-0.34]])  # (2x1)

# Capa de salida
W3 = np.array([[0.8, 0.8]])  # (1x2)
b3 = np.array([[0.7]])  # (1x1)

# Funciones de activacion
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return (x > 0).astype(float)



# Propagacion hacia adelante
def forward(X):
    # Primera capa
    S = W1 @ X.T + b1  # Ajustamos para que las dimensiones sean compatibles
    T = sigmoid(S)
```

```
    # Segunda capa
    U = W2 @ T + b2
    V = relu(U)

    # Capa de salida
    Y_pred = W3 @ V + b3

    return S, T, U, V, Y_pred
```

3.5. Función de Coste

La función de coste utilizada para evaluar el rendimiento de la red neuronal es el error cuadrático medio (MSE), dado por:

$$C = \frac{1}{m} \sum_{i=1}^{m} (V_D - Y_{pred})^2,$$

donde Y_{pred} es la predicción de V_D para los voltajes V_{CC} y V_D es el valor real calculado anteriormente.

```
# Coste (MSE)
def compute_cost(Y, Y_pred):
    m = Y.shape[1]
    cost = np.sum((Y - Y_pred) ** 2) / m
    return cost
```

3.6. Propagación hacia Atrás

Para ajustar los pesos y sesgos de la red, calculamos las derivadas parciales de la función de coste con respecto a cada parámetro, usando el método de retropropagación.

3.6.1. Derivadas en la Capa de Salida

$$\frac{\partial C}{\partial Y_{pred}} = Y - Y_{pred},$$

$$\frac{\partial C}{\partial W_3} = \frac{1}{m} \left(\frac{\partial C}{\partial Y_{pred}} \right) V^T, \quad \frac{\partial C}{\partial b_3} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C}{\partial Y_{pred}}.$$

3.6.2. Derivadas en la Segunda Capa

$$\frac{\partial C}{\partial U} = \left(W_3^T \frac{\partial C}{\partial Y_{pred}} \right) \odot \mathrm{ReLU}'(U),$$

$$\frac{\partial C}{\partial W_2} = \frac{1}{m} \left(\frac{\partial C}{\partial U} \right) T^T, \quad \frac{\partial C}{\partial b_2} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C}{\partial U}.$$

3.6.3. Derivadas en la Primera Capa

$$\frac{\partial C}{\partial S} = \left(W_2^T \frac{\partial C}{\partial U}\right) \odot \sigma'(S),$$

$$\frac{\partial C}{\partial W_1} = \frac{1}{m}\left(\frac{\partial C}{\partial S}\right) X^T, \quad \frac{\partial C}{\partial b_1} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial C}{\partial S}.$$

```
# Propagacion hacia atras (backpropagation)
def backward(X, Y, S, T, U, V, Y_pred):
    m = X.shape[1]

    # Derivada de la capa de salida
    dY_pred = Y_pred - Y  # dC/dY_pred
    dW3 = dY_pred @ V.T / m
    db3 = np.sum(dY_pred, axis=1, keepdims=True) / m

    # Derivadas de la segunda capa
    dV = W3.T @ dY_pred
    dU = dV * relu_derivative(U)
    dW2 = dU @ T.T / m
    db2 = np.sum(dU, axis=1, keepdims=True) / m

    # Derivadas de la primera capa
    dT = W2.T @ dU
    dS = dT * sigmoid_derivative(S)
    dW1 = dS @ X / m
    db1 = np.sum(dS, axis=1, keepdims=True) / m

    return dW1, db1, dW2, db2, dW3, db3
```

3.7. Actualización de Parámetros

Los pesos y sesgos se actualizan en cada iteración de entrenamiento con la regla de gradiente descendente, usando momento y learning rate con los mismos valores que en el problema anterior, 0.5 y 0.01 respectivamente:

$$W_k := W_k - \alpha \frac{\partial C}{\partial W_k}, \quad b_k := b_k - \alpha \frac{\partial C}{\partial b_k},$$

donde α es la tasa de aprendizaje (en este caso 1) y k denota cada capa.

```
# Actualizacion de pesos
def update_parameters(dW1, db1, dW2, db2, dW3, db3, momentum_terms, alpha=0.01,
    global W1, b1, W2, b2, W3, b3

    W1_m, b1_m, W2_m, b2_m, W3_m, b3_m = momentum_terms
```

```
W1_m = momentum * W1_m + (1 - momentum) * dW1
b1_m = momentum * b1_m + (1 - momentum) * db1
W2_m = momentum * W2_m + (1 - momentum) * dW2
b2_m = momentum * b2_m + (1 - momentum) * db2
W3_m = momentum * W3_m + (1 - momentum) * dW3
b3_m = momentum * b3_m + (1 - momentum) * db3

W1 -= alpha * W1_m
b1 -= alpha * b1_m
W2 -= alpha * W2_m
b2 -= alpha * b2_m
W3 -= alpha * W3_m
b3 -= alpha * b3_m
```

3.8. Resultados

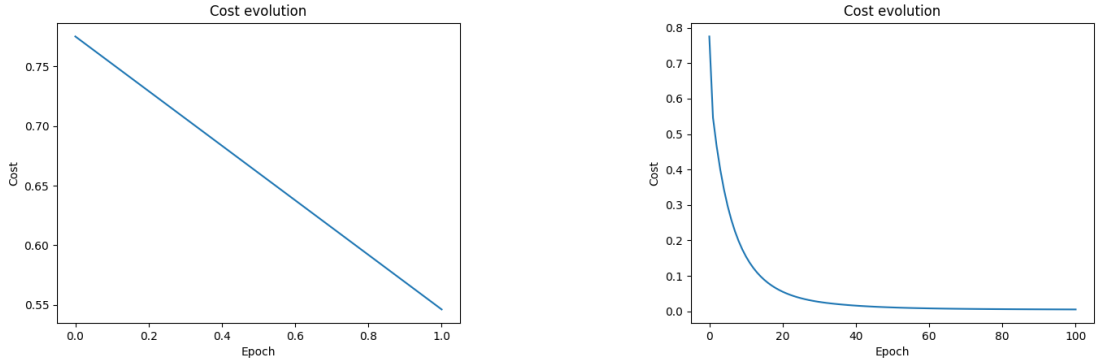
3.8.1. Función lambertw de scipy.special

Tras ejecutar el código expuesto anteriormente, obtenemos los siguientes valores para una época:

$$C = 0{,}5462$$

$$Y_{pred} = \begin{pmatrix} 1{,}3843 & 1{,}3767 & 1{,}3728 \end{pmatrix}$$

Mostramos ahora graficamente la reducción del coste para 1 y 100 épocas:



(a) 1 Época

(b) 100 Épocas

Figura 3.1: Comparativa con diferente cantidad de épocas

Recordando la utilidad inicial de esta práctica, podemos calcular ahora la tensión en la resistencia de forma trivial, siendo la predicción de Y, la tensión del diodo (V_d):

$$I = I_0 \left(e^{\frac{V_d}{\eta \cdot V_T}} - 1 \right)$$

$$V_r = R \cdot I$$

Obtenemos los siguientes valores:

$$V_R = \begin{bmatrix} 1{,}32555468 \times 10^{13} \\ 9{,}88726796 \times 10^{12} \\ 8{,}53305304 \times 10^{12} \end{bmatrix}$$

Estos valores (probablemente muy elevados) están determinados por el reducido número de muestras y épocas para entrenar la red.

3.8.2. Función lambert entrenada

Si usamos la función entrenada en el ejercicio 2 para una sola época, obtenemos los siguientes resultados:

$$\text{C:} \quad 17{,}55438536$$

$$W_s = \begin{bmatrix} 0{,}13679849 & 0{,}21250301 & -0{,}16270926 \end{bmatrix}$$
$$b_s = \begin{bmatrix} 0{,}24167737 & -0{,}09136912 & 0{,}17526973 \end{bmatrix}$$

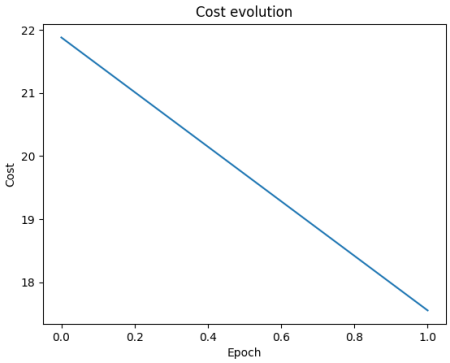
$$W_u = \begin{bmatrix} 0{,}84136333 & -0{,}6 \\ 0{,}74683551 & 0{,}9 \\ 0{,}51386654 & -0{,}6 \end{bmatrix}$$
$$b_u = \begin{bmatrix} 0{,}51399764 & -0{,}34 \end{bmatrix}$$

$$W_y = \begin{bmatrix} 0{,}92700965 \\ 0{,}5 \end{bmatrix}$$
$$b_y = \begin{bmatrix} 0{,}77999704 \end{bmatrix}$$

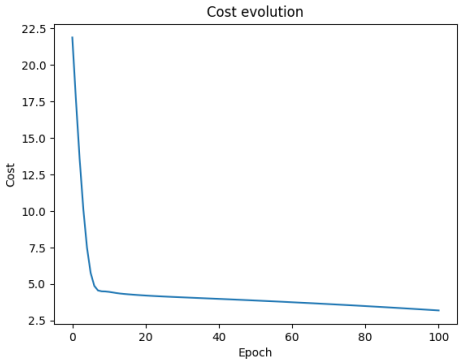
$$V_D = \begin{bmatrix} 2{,}40890774 \\ 2{,}5137368 \\ 2{,}58974416 \end{bmatrix}$$

$$V_R = \begin{bmatrix} 1{,}72787774 \times 10^{30} \\ 9{,}73956039 \times 10^{31} \\ 1{,}81191708 \times 10^{33} \end{bmatrix}$$

Nuevamente, junto con la evolución del coste para diferente número de épocas:



(a) 1 Época



(b) 100 Épocas

Figura 3.2: Comparativa con diferente cantidad de épocas