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

Práctica 2

Entrega grupal

22 de noviembre de 2024

1. Cuestión 1

1.1. Enunciado

Calcular el valor de la función W_0 en tres puntos $x_0 = 0 \cdot e^0, \quad x_1 = 1 \cdot e^1, \quad x_2 = 2 \cdot e^2$.

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. 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.

2.3. 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 igual a 0.01. La actualización de pesos y sesgos se realiza de la siguiente forma:
1. **Cálculo del término de momento**: 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 momento 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.

Recordemos que en la primera época no se usa momento.

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.

2.4. 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`.

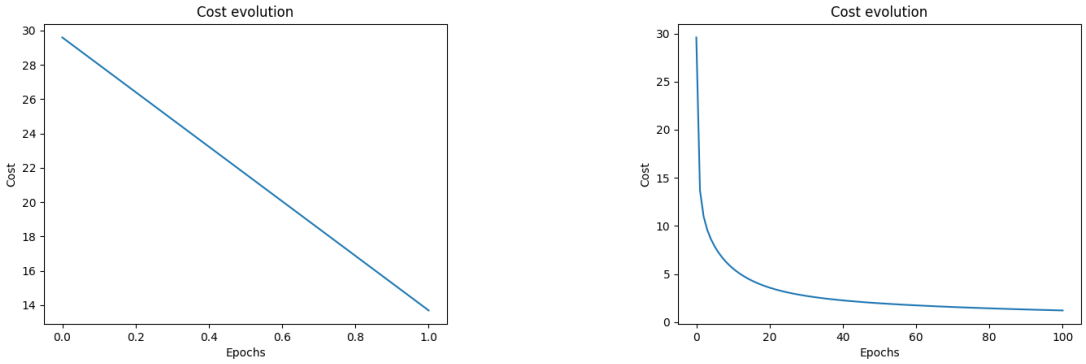
2.5. Resultados

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 inicial:** 29.59559685
- **Coste final:** 13.69094522
- **Pesos y sesgos aprendidos:**
  - $Wu = \begin{bmatrix} 0{,}08159339 & -0{,}68622434 & -0{,}11742454 \end{bmatrix}$
  - $bu = \begin{bmatrix} 0{,}27116171 & -0{,}37917983 & -0{,}00759286 \end{bmatrix}$
  - $Wy = \begin{bmatrix} 1{,}32598667 \\ 7{,}76211242 \\ 3{,}33274683 \end{bmatrix}$
  - $by = \begin{bmatrix} 0{,}39333115 \end{bmatrix}$
- **Predicciones finales:** $\begin{bmatrix} 5{,}95965231 & 3{,}3567872 & 1{,}96943647 \end{bmatrix}$

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 su disminución (incluyendo momento a partir de la segunda época).



(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. Y la librería tensorflow para calcular los gradientes.
- **Aproximación 2:** En este caso, creamos una función `lambert` basada en la red neuronal entrenada previamente, que fue definida en el archivo `P2E2.py`. 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`. Los gradientes se calculan siguiendo un método matricial implementado manualmente.

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. Aproximación 1

Usando la función `lambertw` de la librería `scipy.special` de esta forma:

```python
from scipy.special import lambertw
import numpy as np

def calc_vd(VCC):
    I0 = 1e-12
    n = 1
    VT = 0.026
    R = 100

    argumento = (R * I0) / (n * VT)
    exponente = (VCC + R * I0) / (n * VT)

    W0 = lambertw(argumento * np.exp(exponente)).real

    VD = VCC + (R * I0) - (n * VT * W0)

    return VD
```

3.3.1. Estructura de la red

Se muestra ahora la creación de la red con tensorflow (consultar el código para más detalle):

```python
model = tf.keras.Sequential(
    [
        # Capa 1: Densa lineal sigmoide
        tf.keras.layers.Dense(
            3,
            activation="sigmoid",
            input_shape=(1,),
            kernel_initializer=tf.constant_initializer([[0.05], [0.15], [-0.2]]),
            bias_initializer=tf.constant_initializer([0.23, -0.1, 0.17]),
        ),
        # Capa 2: Densa lineal relu
        tf.keras.layers.Dense(
            2,
            activation="relu",
            kernel_initializer=tf.constant_initializer(
                [[0.8, -0.6], [0.7, 0.9], [0.5, -0.6]]
            ),
            bias_initializer=tf.constant_initializer([0.45, -0.34]),
        ),
        # Capa 3: Densa lineal que produce un vector de salida
        tf.keras.layers.Dense(
            1,
            activation=None,
            kernel_initializer=tf.constant_initializer([[0.8], [0.5]]),
            bias_initializer=tf.constant_initializer([0.7]),
        ),
    ]
)

model.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.01), loss="mse", metrics=["mse"]
)

```

3.3.2. Resultados

- **Predicciones iniciales:** $\begin{bmatrix} 1{,}9262 \\ 1{,}9545 \\ 1{,}9856 \end{bmatrix}$
- **Error inicial (MSE):** 1.7328434
- **Pesos y sesgos aprendidos:**
  - $Ws = \begin{bmatrix} 0{,}02672244 & 0{,}13242216 & -0{,}21094187 \end{bmatrix}$
  - $bs = \begin{bmatrix} 0{,}22608304 & -0{,}10309005 & 0{,}16800411 \end{bmatrix}$
  - $Wu = \begin{bmatrix} 0{,}78674686 & -0{,}6 \\ 0{,}6855674 & 0{,}9 \\ 0{,}49425203 & -0{,}6 \end{bmatrix}$
  - $bu = \begin{bmatrix} 0{,}42893875 & -0{,}34 \end{bmatrix}$
  - $Wy = \begin{bmatrix} 0{,}7586789 & 0{,}5 \end{bmatrix}$

- $by = \begin{bmatrix} 0{,}67367345 \end{bmatrix}$

- **Predicciones finales:** $\begin{bmatrix} 1{,}7856 \\ 1{,}7949 \\ 1{,}8115 \end{bmatrix}$
- **Error final (MSE):** 1.3414707

3.4. Aproximación 2

Emplearemos ahora una función lambert que usa la red entrenada en el ejercicio 2 (el modelo fue guardado en formato .pkl, con los pesos y biases mostrados en el ejercicio anterior):

```
def lambert(X):
    # Load weights and biases
    with open("lambert_model.pkl", "rb") as f:
        weights = pickle.load(f)
    Wu, bu, Wy, by = weights["Wu"], weights["bu"], weights["Wy"], weights["by"]

    layer1 = X @ Wu + np.ones((X.shape[0], 1)) @ bu
    layer2 = sigmoid(layer1)
    layer3 = layer2 @ Wy + np.ones((X.shape[0], 1)) @ by

    return layer3
```

3.4.1. Ejercicio 3 con la función lambert entrenada:

Para calcular V_D iniciales se usa la función lambert anteriormente descrita:

```
# ...

def main():
    # Getting VD using the Lambert function from P2E2
    IO = 1e-12
    ETA = 1
    VT = 0.026
    R = 100

    VCC = np.array([[3], [6], [9]])

    WO = lambert(R * IO / (ETA * VT) * e ** ((VCC + R * IO) / (ETA * VT))).real

    VD = VCC + R * IO - ETA * VT * WO

    # ...

```

3.4.2. Actualización de los pesos con enfoque matricial

Se muestra como se actualizan los pesos para esta segunda aproximación:

```
# ...

V_5 = Ws
V_4 = bs
V_3 = Wu
V_2 = bu
V_1 = Wy
V0 = by
V1 = X @ V_5 + np.ones((m, 1)) @ V_4
V2 = sigmoid(V1)
V3 = V2 @ V_3 + np.ones((m, 1)) @ V_2
V4 = ReLU(V3)
V5 = V4 @ V_1 + np.ones((m, 1)) @ V0
V6 = (1 / m) * ((Y - V5).T @ (Y - V5))
fs[i] = V6

# Backward pass
# V6_ = 1
V5_ = -2 / m * (Y - V5)
V4_ = V5_ @ V_1.T
V3_ = V4_ * (V3 > 0)  # ReLU derivative
V2_ = V3_ @ V_3.T
V1_ = V2_ * V2 * (np.ones((V2.shape[0], V2.shape[1])) - V2)  # Sigmoid derivative
V0_ = np.ones((m, 1)).T @ V5_
V_1_ = V4.T @ V5_
V_2_ = np.ones((m, 1)).T @ V3_
V_3_ = V2.T @ V3_
V_4_ = np.ones((m, 1)).T @ V1_
V_5_ = X.T @ V1_

# First epoch momentum = 0
Ws_m = momentum * Ws_m + (1 - momentum) * V_5_
bs_m = momentum * bs_m + (1 - momentum) * V_4_
Wu_m = momentum * Wu_m + (1 - momentum) * V_3_
bu_m = momentum * bu_m + (1 - momentum) * V_2_
Wy_m = momentum * Wy_m + (1 - momentum) * V_1_
by_m = momentum * by_m + (1 - momentum) * V0_

Ws -= lr * Ws_m
bs -= lr * bs_m
Wu -= lr * Wu_m
bu -= lr * bu_m
Wy -= lr * Wy_m
by -= lr * by_m

# ...
```

3.4.3. Resultados

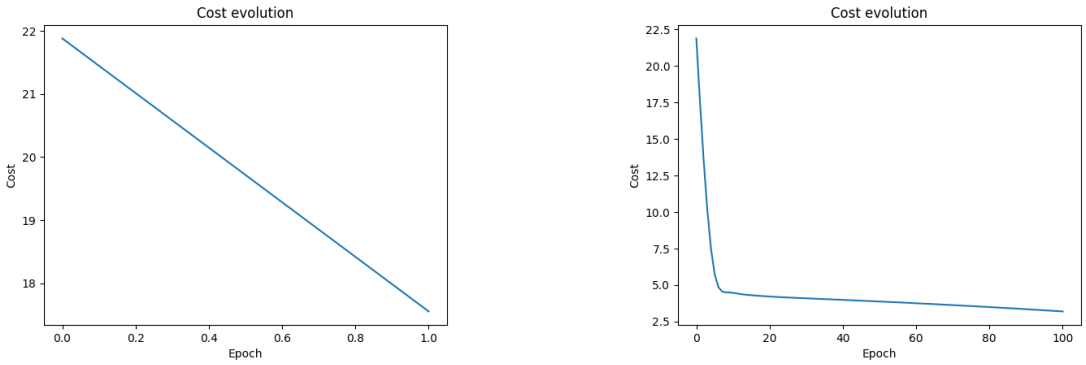
- **Predicciones iniciales:** $\begin{bmatrix} 1{,}9262 \\ 1{,}9545 \\ 1{,}9856 \end{bmatrix}$

- **Error inicial (MSE):** 21.88065055
- **Pesos y sesgos aprendidos:**
  - $Ws = \begin{bmatrix} 0{,}13679849 & 0{,}21250301 & -0{,}16270926 \end{bmatrix}$
  - $bs = \begin{bmatrix} 0{,}24167737 & -0{,}09136912 & 0{,}17526973 \end{bmatrix}$
  - $Wu = \begin{bmatrix} 0{,}84136333 & -0{,}6 \\ 0{,}74683551 & 0{,}9 \\ 0{,}51386654 & -0{,}6 \end{bmatrix}$
  - $bu = \begin{bmatrix} 0{,}51399764 & -0{,}34 \end{bmatrix}$
  - $Wy = \begin{bmatrix} 0{,}92700965 & 0{,}5 \end{bmatrix}$
  - $by = \begin{bmatrix} 0{,}77999704 \end{bmatrix}$
- **Predicciones finales:** $\begin{bmatrix} 2,40890774 \\ 2,5137368 \\ 2,58974416 \end{bmatrix}$
- **Error final (MSE):** 17,55438536

Con estas predicciones de V_D, aplicamos la fórmula y obtenemos los siguientes valores de V_R:

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

Mostramos ahora la evolución del coste para diferente número de épocas (learning rate = 0.01 y momento = 0.5):



(a) 1 Época (b) 100 Épocas

Figura 3.1: Comparativa con diferente cantidad de épocas

3.5. Conclusión

Ambas aproximaciones buscan resolver el mismo problema de predicción del voltaje V_D utilizando una red neuronal, pero con enfoques distintos en la resolución de la función de Lambert y el cálculo de gradientes:

- Aproximación 1 (scipy lambertw):
  - Utiliza la función lambertw de la biblioteca scipy.special, que proporciona una solución numérica directa y eficiente para la ecuación de Lambert.
  - Emplea TensorFlow para entrenar la red neuronal y calcular automáticamente los gradientes mediante diferenciación automática en modo reverse.
  - Presenta una mayor precisión en los resultados debido al uso de la función de Lambert predefinida, con un error inicial más bajo (MSE: 1.7328) y una convergencia más eficiente hacia el objetivo (MSE final: 1.3415).
- Aproximación 2 (Red entrenada):
  - Reemplaza la función de Lambert por la red neuronal entrenada en el ejercicio 2.
  - Los gradientes se calculan siguiendo un enfoque matricial.
  - Mayor error inicial (MSE: 21.8807) y final (MSE: 17.5544).

La aproximación 1 es más precisa debido a la solución numérica directa de la función de Lambert, mientras que la aproximación 2 tiene un margen de error mayor por depender del entrenamiento de la red, la cual solo usa 3 muestras y una época de entrenamiento.

Apéndice

El código relacionado con este documento está disponible en el siguiente enlace de GitHub:

Repositorio