SISTEMAS INTELIGENTES I

Práctica 1: Análisis y Diseño de Arquitecturas Neuronales Supervisadas para la Clasificación de Patrones (Backpropagation)

2020/21

27 de diciembre de 2020

José Amusquívar Poppe | Prashant Jeswani Tejwani

Universidad de Las Palmas de Gran Canaria

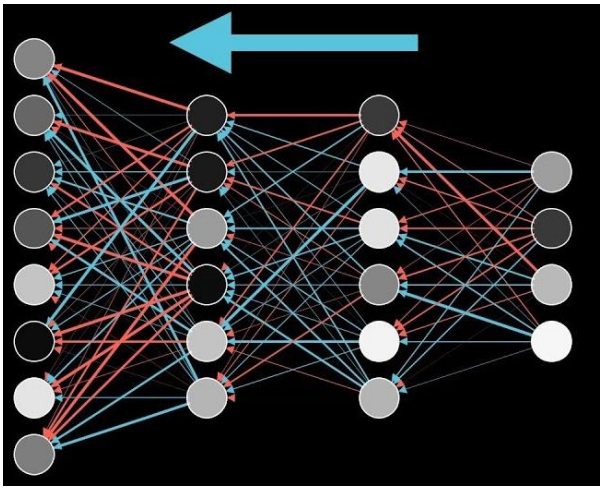
Escuela de Ingeniería en Informática

Índice

Estudio y análisis de la arquitectura BPNN

Cuando necesitamos representar problemas complejos, no nos basta únicamente con un simple perceptrón, sino que necesitamos una red de perceptrones interconectados entre ellos.

Para el entrenamiento de una red se debe tener en cuenta que la salida de cada neurona no va a depender únicamente de las entradas del problema, sino que también depende de las salidas que ofrezcan el resto de las neuronas. Por este mismo motivo también podemos afirmar que el error cometido por una neurona no solo va a depender de que sus pesos sean los correctos o no, sino que dependerá del error que traiga acumulado del resto de neuronas que le precedan en la red.



Por lo que necesitamos un algoritmo eficiente que nos permita adaptar todos los pesos de una red multicapa, no sólo los de la capa de salida.

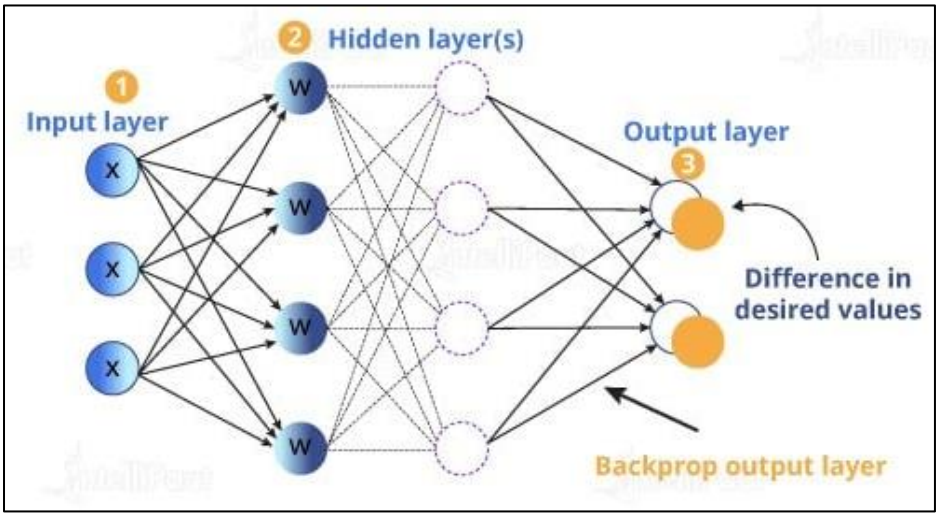
La **arquitectura de *Backpropagation*** consiste en interconectar varias unidades de procesamiento en capas (las neuronas de cada capa no se interconectan entre sí). Sin embargo, cada neurona de una capa proporciona una entrada a cada una de las neuronas de la siguiente capa, esto es, cada neurona transmitirá su señal de salida a cada neurona de la capa siguiente. Es decir, una red neuronal de retro propagación es una red neuronal multicapa de retroalimentación que consta de una capa de entrada, una (o varias) capa oculta y una capa de salida.

Proceso de aprendizaje de la BPNN

Las redes *Backpropagation* tienen un método de entrenamiento supervisado. El objetivo de la retro propagación es optimizar los pesos (los cuales normalmente son inicializados aleatorios) para que la red neuronal pueda aprender a asignar correctamente entradas arbitrarias a salidas. El algoritmo conlleva una fase de propagación hacia adelante y otra fase de propagación hacia atrás.

Propagación hacia adelante: Esta fase de propagación hacia adelante se inicia cuando se presenta un conjunto de datos en la capa de entrada de la red. Las unidades de entrada toman el valor de su correspondiente elemento de entrada y se calcula el valor de activación (es necesario que la función de activación sea derivable). A continuación, las demás capas realizarán la fase de propagación hacia adelante que determina el nivel de activación de las otras capas hasta obtener la salida de la red (predicción de la red). Se realiza el cálculo del error cometido por la red mediante una función de pérdida.

Propagación hacia atrás: Una vez se ha completado la fase de propagación hacia adelante se inicia la fase de corrección o fase de propagación hacia atrás. Los cálculos de las modificaciones de todos los pesos de las conexiones empiezan por la capa de salida y continua hacia atrás a través de todas las capas de la red hasta la capa de entrada. El objetivo con la propagación hacia atrás es actualizar cada uno de los pesos en la red para que provoquen que la salida real esté más cerca de la salida objetivo, minimizando así el error para cada neurona de salida y la red como un todo.



Cabe destacar que algunas redes *Backpropagation* utilizan unidades llamadas *bias* como parte de cualquiera de las capas ocultas y de la capa de salida. Estas unidades presentan constantemente un nivel de activación de valor 1. Además, esta unidad está conectada a todas las unidades de la capa inmediatamente superior y los pesos asociados a dichas conexiones son ajustables en el proceso de entrenamiento. La utilización de esta unidad tiene un doble objetivo: mejorar las propiedades de convergencia de la red y ofrecer un nuevo efecto umbral sobre la unidad que opera.

Análisis del conjunto de datos

Se ha hecho uso del conjunto de datos: **Organización y selección de Jornadas de Conducción adecuadas en el Transporte de Mercancías Peligrosas**.

El conjunto de datos se compone de las siguientes características:

Fecha y hora de la observación
Longitud del Vehículo
Carril de circulación
Velocidad de circulación
Peso del Vehículo
Número de ejes
Temperatura del aire
Humedad relativa
Tipo de precipitación
Intensidad de la precipitación
Dirección del viento
Velocidad del viento
Estado carretero
Accidente (SI/NO)

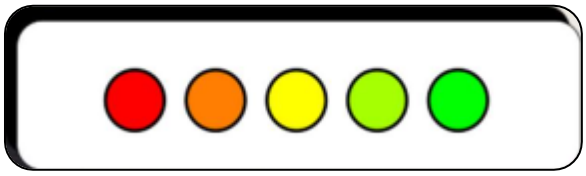
Antes de pasar el conjunto de datos a los modelos se ha decidido preprocesarlos para eliminar datos inconsistentes u observaciones sin valores:

Primeramente, se ha modificado la columna de la etiqueta a predecir (Accidente) estableciendo un 1 si se ha producido un accidente o 0 para lo contrario.

Luego, se ha realizado una codificación one-hot (*one-hot encoding)* a las columnas no numéricas: tipo de precipitación, intensidad de precipitación y estado de carretera.

Finalmente, la columna de la fecha y hora se ha optado por preservar el mes, día y hora, por lo que se han añadido tres nuevas columnas para los respectivos datos.

La salida de las redes será un valor que expresará el porcentaje de tener o no un accidente dadas las características. Dependiendo del porcentaje obtenido, el resultado se reducirá en 5 tipos de señales:

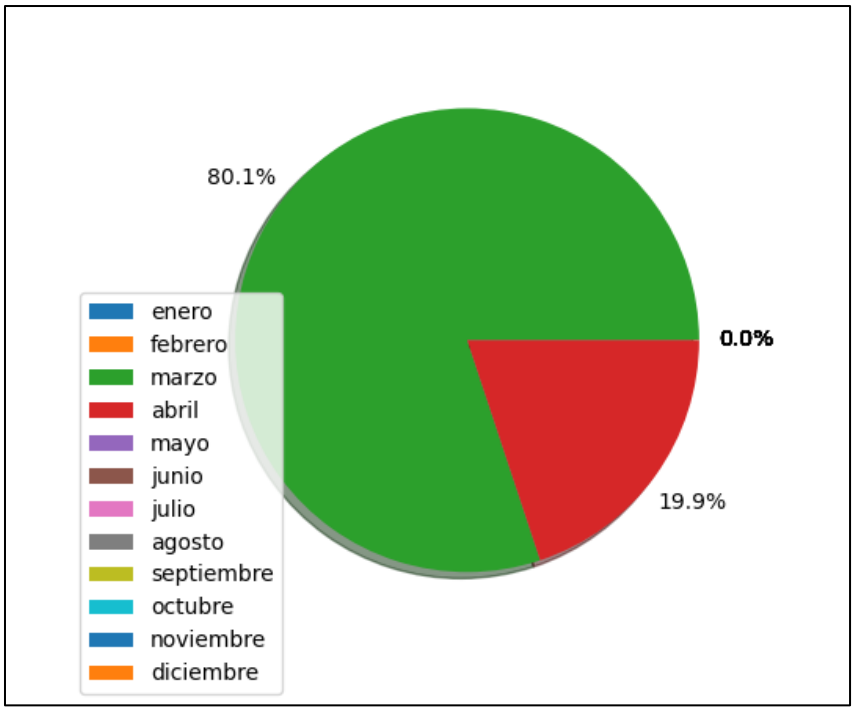


Siendo el rojo (76% - 100%), naranja (51% - 75%), amarillo (41% - 50%), verde claro (21% - 40%) y verde oscuro (0% – 20%) probabilidades de que haya un accidente. Para replicar el semáforo se ha optado por representarlo en la consola de la siguiente forma:

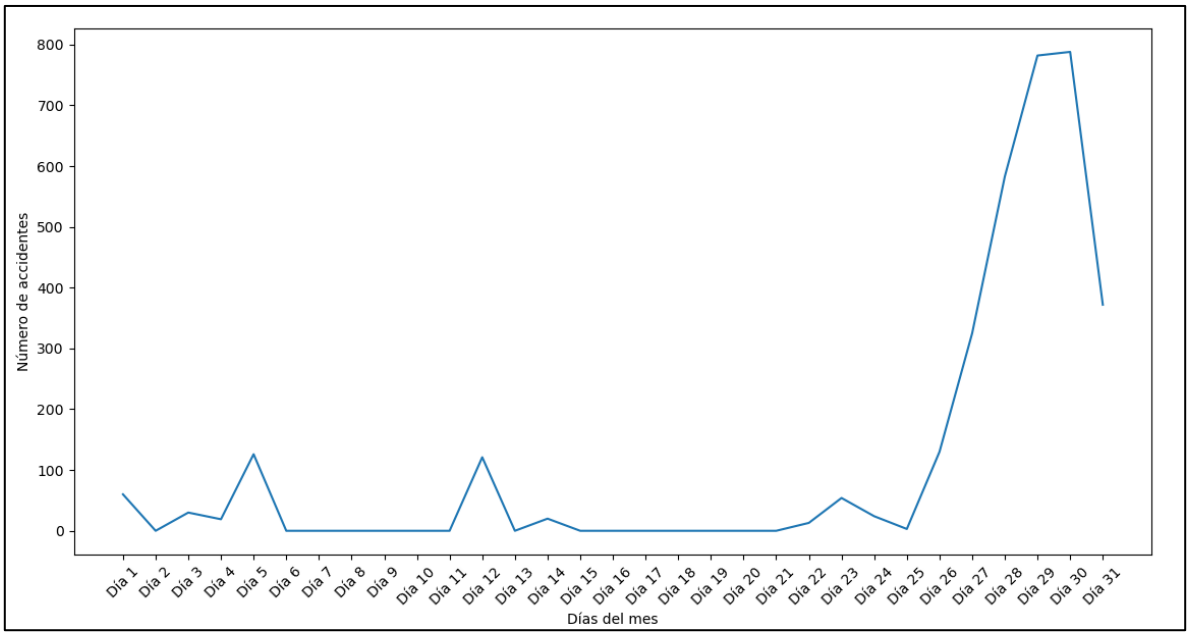
```
Original:  0  ---- Predicted:  [0.70132995] -------------------- Not match ---------
Original:  0  ---- Predicted:  [0.] --------- |0|0|0|0|0| ------- VERY GOOD ---------
Original:  0  ---- Predicted:  [0.] --------- |0|0|0|0|0| ------- VERY GOOD ---------
Original:  1  ---- Predicted:  [0.85525227] --------- |0|0|0|0|0| ------- VERY BAD ---------
Original:  0  ---- Predicted:  [9.267599e-15] --------- |0|0|0|0|0| ------- VERY GOOD ---------
Original:  0  ---- Predicted:  [3.9985076e-38] --------- |0|0|0|0|0| ------- VERY GOOD ---------
Original:  1  ---- Predicted:  [0.7859157] --------- |0|0|0|0|0| ------- VERY BAD ---------
Original:  1  ---- Predicted:  [0.6020503] --------- |0|0|0|0|0| ------- BAD ---------
Original:  0  ---- Predicted:  [0.] --------- |0|0|0|0|0| ------- VERY GOOD ---------
Original:  1  ---- Predicted:  [0.491485] -------------------- Not match ---------
```

Se imprime las etiquetas originales, la predicción y finalmente el semáforo con su color correspondiente dependiendo de la probabilidad obtenida para la observación.

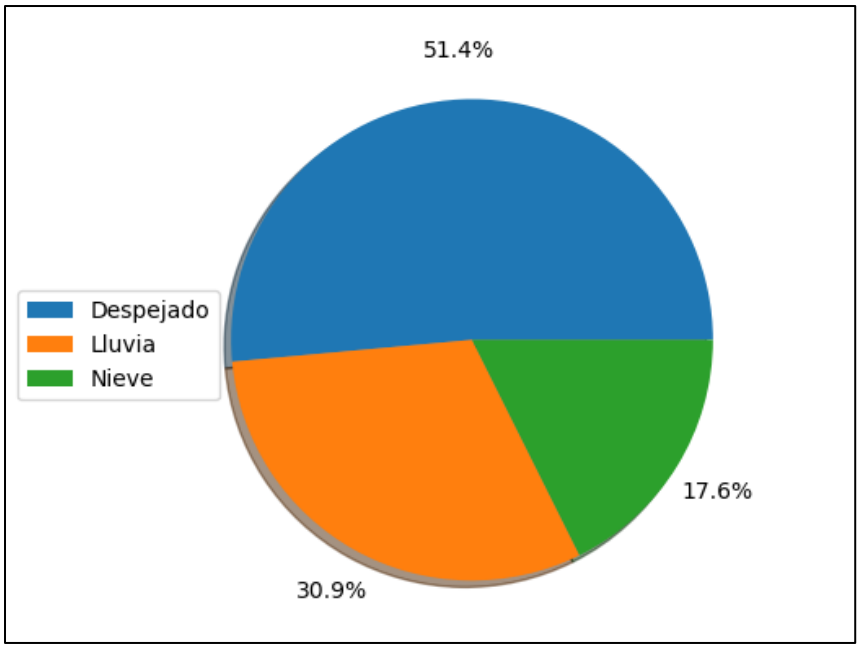
Para obtener un primer contacto con los datos, representamos los datos según algunas características, por ejemplo, en qué mes ha habido más accidentes:



Podemos ver que en primavera se producen más accidentes del año. Sería interesante visualizar en qué días del mes ha habido más accidentes:



Obtenemos que a finales de mes es donde más accidentes se producen, a partir del día 25 – día 31. Finalmente, si representamos en qué condiciones climatológicas se producen más accidentes:



Observamos que la mayoría de los accidentes se producen con cielos despejados, al cual le sigue en condiciones de lluvia y finalmente en condiciones de nieve.

Modelo 1

Para el modelo 1 (sin Keras) se han establecido los siguientes parámetros e hiperparámetros:

```
model = BackPropagation(0.001, 50, 1, True, 40)

model.fit(x_train, y_train, x_valid, y_valid, 6, np.array([512, 256, 128, 64, 32, 16]))
```

Modelo 2

Para el modelo 2 (con Keras) se ha optado la misma configuración que en el modelo anterior, aunque se ha optado. Además, se ha escogido el optimizador Adam.

```
# Create model
model = Sequential()
model.add(Dense(512, input_dim=23, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation="sigmoid"))

# Compile and fit model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(X, Y, epochs=50, validation_split=0.2, shuffle=True, batch_size=40)
```

Además, las funciones de activación utilizadas en todas las capas es la función relu menos en la última que será una función sigmoide para obtener un valor entre 0 - 1.

Comparativa de los modelos

Se ha decidido dividir aleatoriamente el conjunto de datos en un conjunto de entrenamiento (95% del conjunto de datos) y conjunto de test la cual la red nunca ha visto (5% del conjunto de datos).

Si comparamos ambos modelos, vemos que el *accuracy* para el modelo de Keras es mayor que el modelo 1, además de tener una pérdida menor. La función de pérdida utilizada es el *binary_crossentropy* ya que es adecuada para valores de salidas que sean probabilidades.

```
Epoch  50  of  50 -----------------
Loss training:  0.20965135312127542
Loss validation:  0.2049499882042078
Training accuracy:  0.9461606790367153
Validation accuracy:  0.9478
```

Figura 1. Métricas del modelo 1

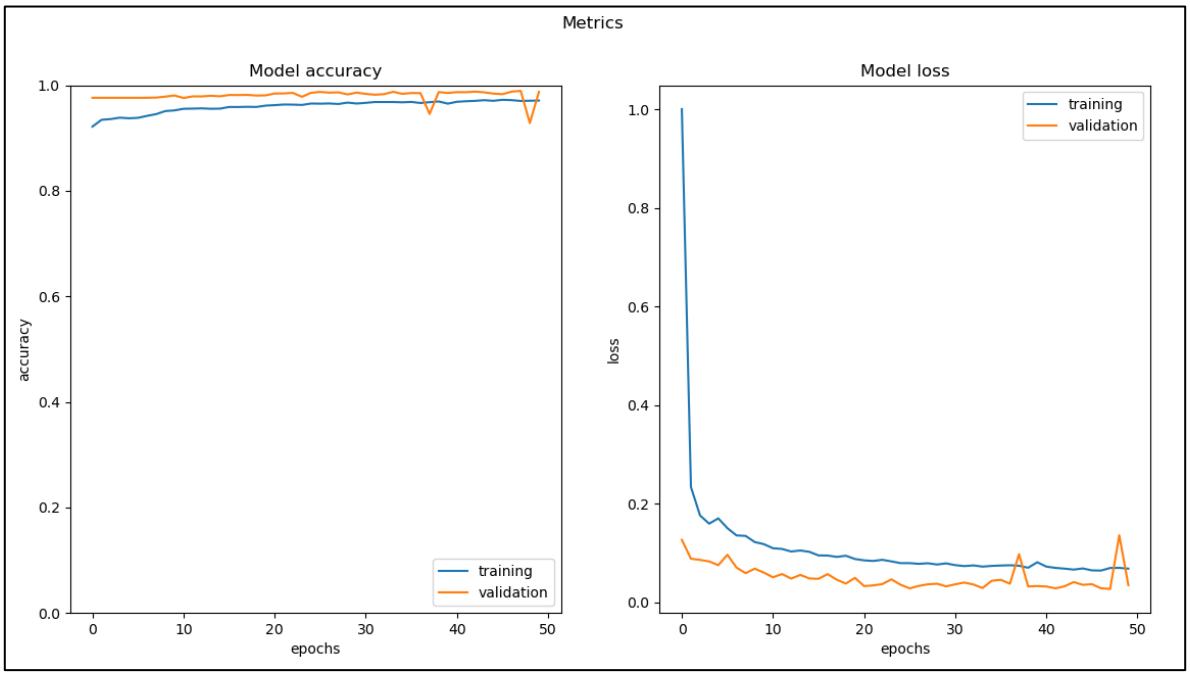
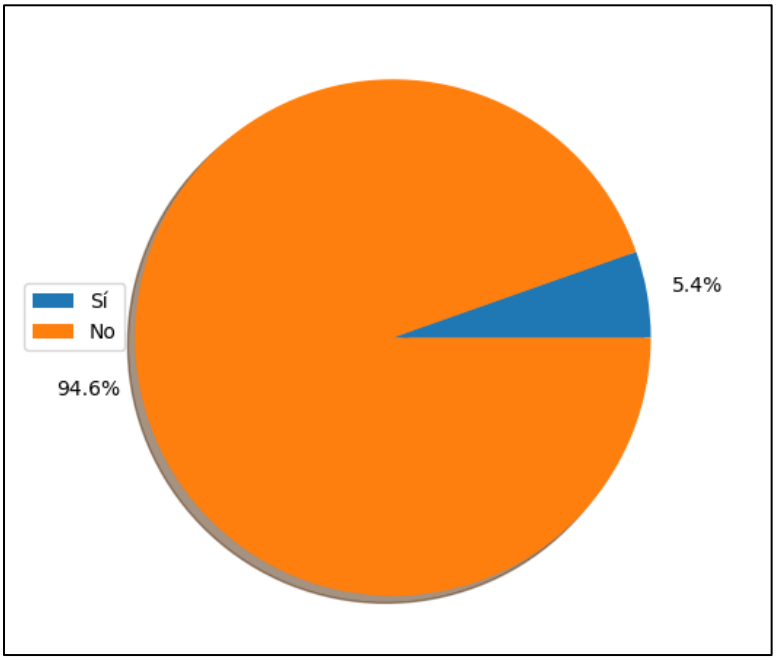


Figura 2. Métricas del modelo 2 (con Keras)

```
Loss - Accuracy -->:  [0.05605127289891243, 0.9771875143051147]
```

Para el *accuracy* observamos que para el modelo 1 es de 94,78% para el conjunto de validación y para los *tests* un 94% y para el modelo 2 (con Keras) es de 98,8% y 97,7% respectivamente. Igual ocurre con la función de pérdida, el error para ambos modelos es bajo, pero el modelo de Keras es mejor.

Pero hemos comprobado que para observaciones las cuales el accidente es igual a 1, el modelo 1 no predice correctamente mientras que para las observaciones donde el accidente es igual a 0 sí. Esto es debido a que, si representamos los datos, observamos que el conjunto de datos está desequilibrado:



Si nos damos cuenta, 94,6% del conjunto de datos son observaciones en donde no ocurre accidente y concuerda con el *accuracy* que se ha obtenido en el modelo 1, es decir, que para las observaciones en donde no se producen accidentes el modelo 1 predice correctamente pero cuando sí se produce accidente el modelo no es capaz de predecir debido a los insuficientes casos en el conjunto de datos.

Por lo que para la red del modelo 1 se necesitaría un conjunto de datos mejor equilibrado (con más casos de accidentes) o tener dos maneras distintas de actualizar el error dándole más peso a las observaciones las cuales se producen accidentes, es decir, ponderar la función de pérdida. Mientras que en el modelo 2 (con Keras) esto no es necesario, esto puede ser debido a que Keras utiliza un optimizador. Además, Keras permite los pesos de las clases con el parámetro *class_weights* si el conjunto de datos está desequilibrado.

Debido a esto, hemos probado utilizar el modelo 1 para otro conjunto de datos para asegurar que el código implementado es correcto. Este conjunto de datos consiste en tener como datos de entrada dos números y como salida la suma de ambos números.

```
items = np.array([[random() / 2 for _ in range(2)] for _ in range(1000)])
targets = np.array([[i[0] + i[1]] for i in items])
```

Se ha aplicado los siguientes cambios en la red: la taza de aprendizaje se ha establecido a 0,2 con 100 épocas y con un tamaño de *batch* igual a 1:

```
model = BackPropagation(0.2, 100, 1, True, 1)

model.fit(x_train, y_train, x_valid, y_valid, 1, np.array([10]))
```

Después de 50 épocas la métrica de pérdida obtenida es:

```
Epoch  100  of  100 ----------------
Loss training:  0.0002672906633823399
Loss validation:  0.0002347348594166335
```

Para comprobar si la red ha entrenado correctamente, se pasan como entrada dos nuevas observaciones:

```
input_ = np.array([[0.3, 0.1], [0.2, 0.7]])
target = np.array([[0.4], [0.9]])

y_predict = model.predict(input_)
```

```
Predicted:  [0.3930073]  Real:  [0.4]
Predicted:  [0.86406522]  Real:  [0.9]
```

Se adjunta los códigos de los modelos junto al informe. Adicionalmente, se adjunta el enlace al repositorio GitHub en el apartado de Referencias.

Referencias

Github. (s.f.). Obtenido de https://github.com/Prashant-JT/Backpropagation