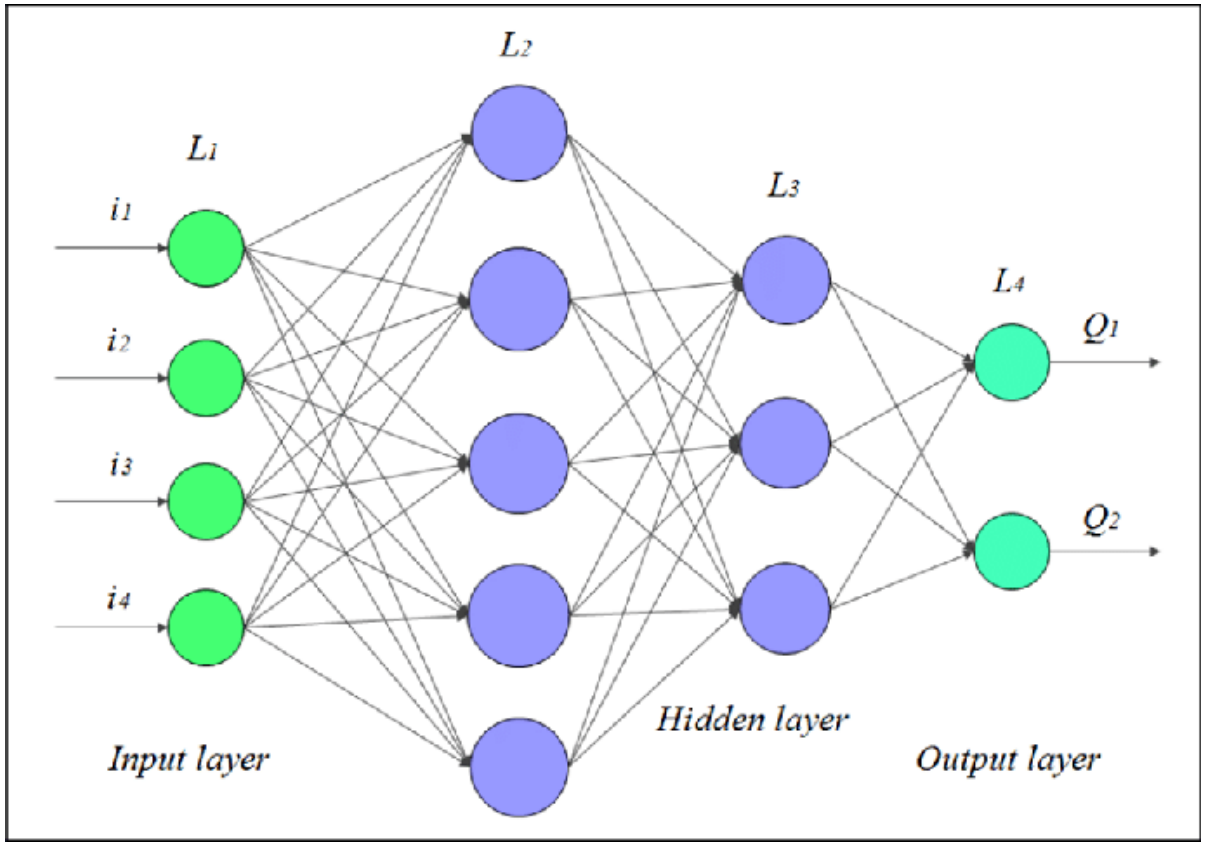
Redes Neuronales Artificiales Implementación teórica

Implementación teórica de una red neuronal artificial usando la función sigmoide como función de activación para todas las capas de la red, y el error cuadrático medio como función de coste (siendo la función de pérdida el error cuadrático)

Nos basaremos en:

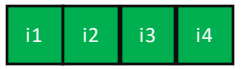


Una red neuronal artificial con esta estructura. Tiene **4 capas.**

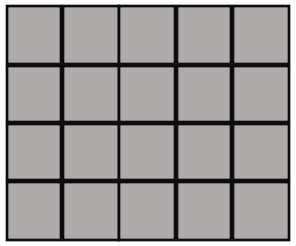
Las capas ocultas y la capa de salida tendrán como función de activación, la sigmoide

Representación red matricial:

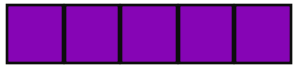
AL(L1) 4



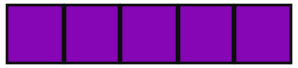
W(L1,L2) 4x5



ZL(L2) 5

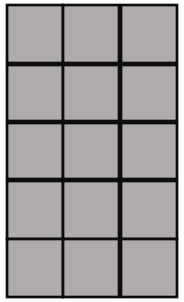


AL(L2) 5



B(L2) 5

W(L2,L3) 5x3

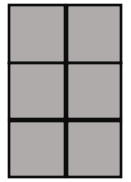


ZL(L3) 3

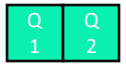
AL(L3) 3

B(L3) 3

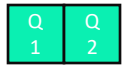
W(L3,L4) 3x2



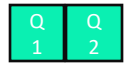
ZL(L4) 2

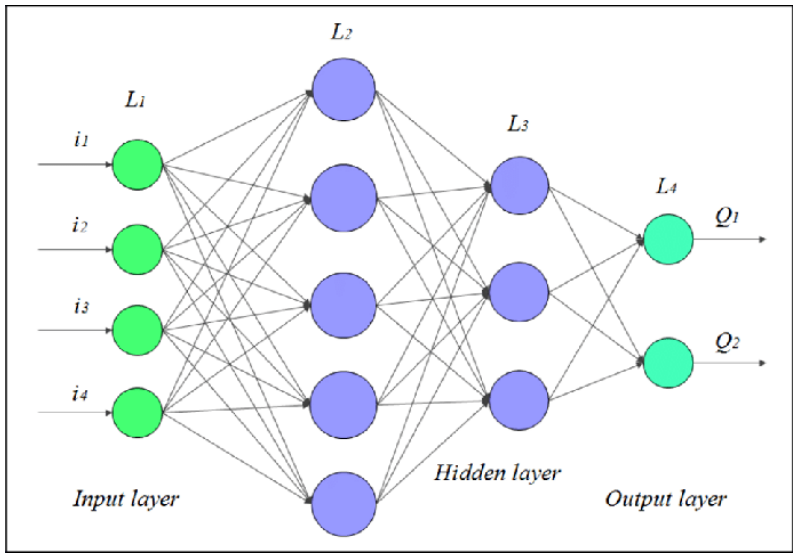


AL(L4) 2



B(L4) 2





B: vector donde se guardan los valores del bias de la neurona en la capa actual. Se suele inicializar con todos sus valores iguales a 0

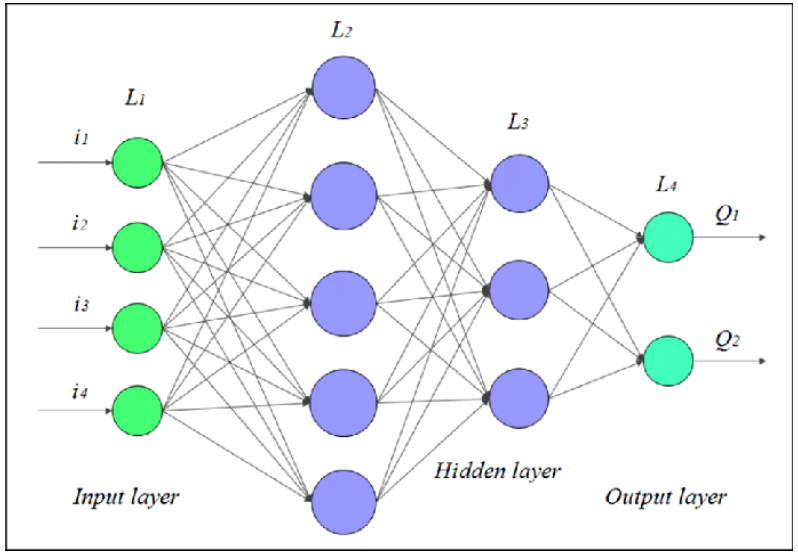
W: matriz con los valores de los pesos de las aristas. Se inicializa con valores aleatorios, hay distintas formas de inicializarlos, pero supongamos números entre 3.0 y –3.0

ZL: vector donde se almacena o los valores de entrada (sólo primera capa) o el resultado para cada neurona de la capa actual el calcular el sumatorio del valor de cada neurona de la capa anterior por el peso de la arista que conecta la neurona de la capa actual, más el bias de dicha neurona (resto de capas). No importan sus valores iniciales

AL: vector donde se almacena el resultado de aplicar a cada elemento en ZL la función de activación de dicha capa. No importan sus valores iniciales

Representación red como listas

```
Red.numero_capas = 4
Red.zl = [
   [ vector 5 elementos sin importar sus valores ],
   [ vector 3 elementos sin importar sus valores ],
   [ vector 2 elementos sin importar sus valores ]
]
Red.al = [
   [ vector 4 elementos sin importar sus valores ],
   [ vector 5 elementos sin importar sus valores ],
   [ vector 3 elementos sin importar sus valores ],
   [ vector 2 elementos sin importar sus valores ]
]
Red.bias = [
   [ vector 5 elementos todos sus valores 0 ],
   [ vector 3 elementos todos sus valores 0 ],
   [ vector 2 elementos todos sus valores 0 ]
]
Red.pesos = [
   [ matriz 4x5 elementos todos sus valores aleatorios entre (3.0, -3.0) ],
   [ matriz 5x3 elementos todos sus valores aleatorios entre (3.0, -3.0) ],
   [ matriz 3x2 elementos todos sus valores aleatorios entre (3.0, -3.0) ]
]
```



Propagación hacia detrás

Acción que permite a la red pasar unos valores de entrada que deseamos evaluar en su extremo izquierdo y, en su extremo derecho, la capa de salida, esperamos recibir unos valores concretos en cada uno de los nodos de esta capa de salida

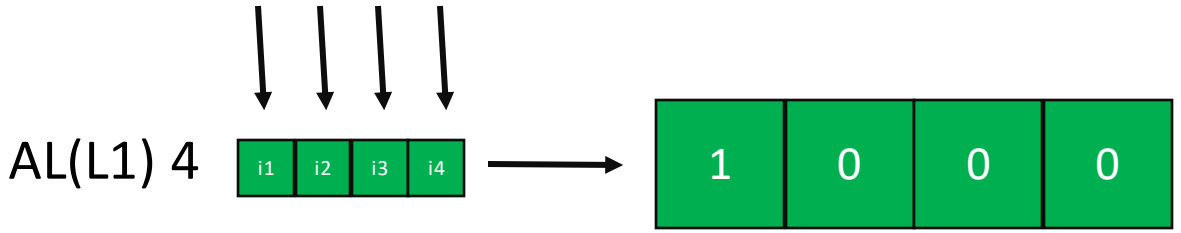
Recordemos que la red tiene 4 capas, aunque se consideran como neuronas solamente los nodos de las capas intermedias (ocultas) y la capa a la derecha (salida), ya que los nodos de la capa a la izquierda (entrada) no modifican los valores de entrada

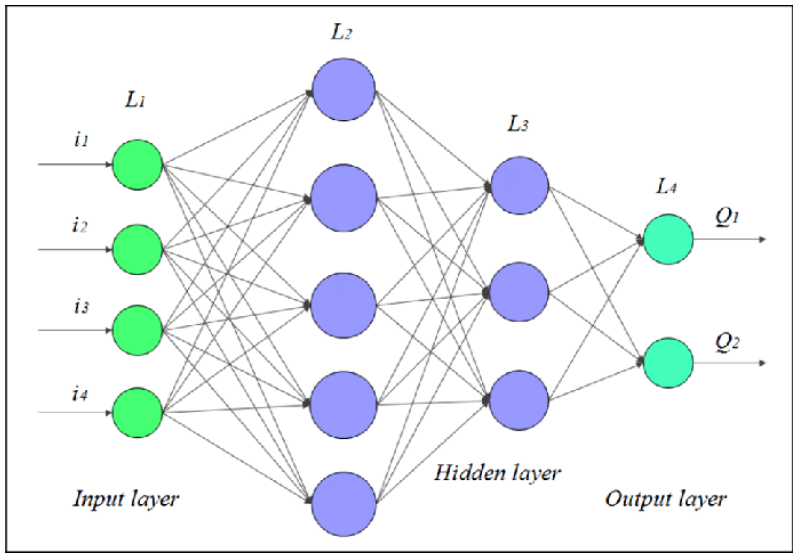
Aparte, recordar que las capas que se consideran neuronas aplican a los resultados que computan una función concreta, llamada función de activación, y nosotros consideraremos que dicha función es la sigmoide. Esta función solamente toma un valor como entrada y lo modifica realizando esta operación:

sigmoid(x):
  devolver $1 / (1 + e^{-x})$

Propagación hacia delante

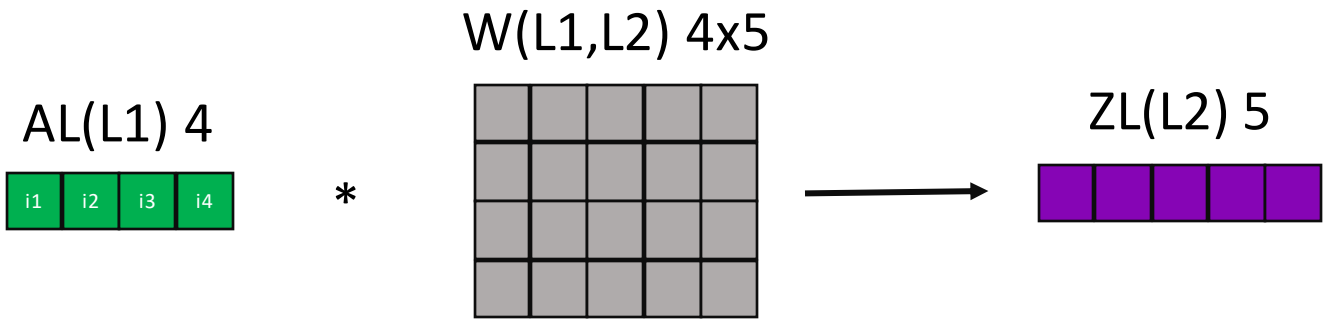
VALORES DE ENTRADA(LISTA) = [1, 0, 0, 0]

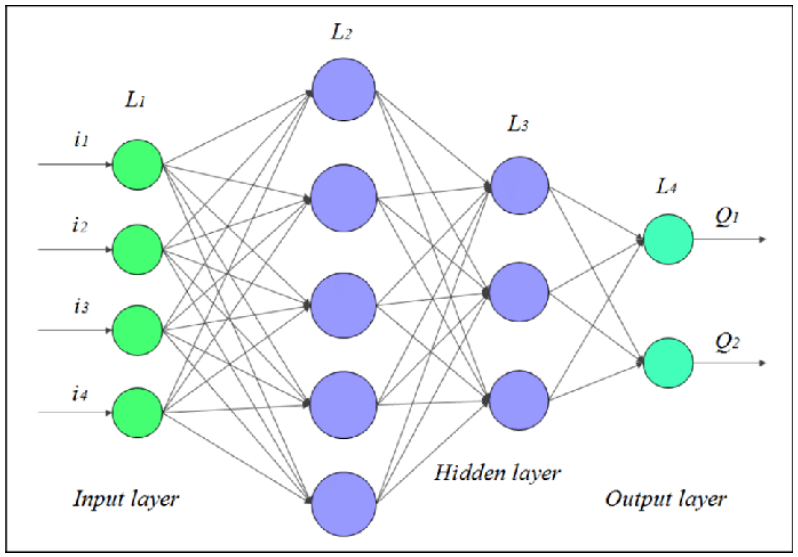




Pasamos los valores de entrada deseados a la primera lista al de la red

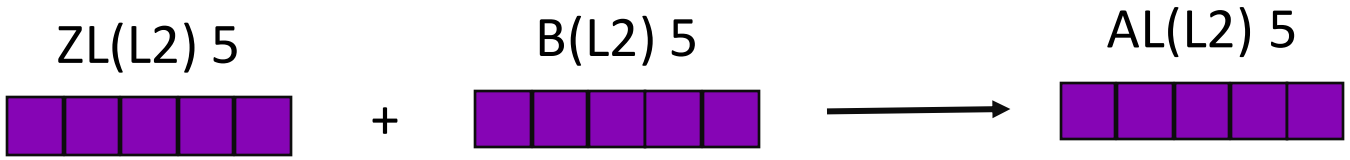
Propagación hacia delante

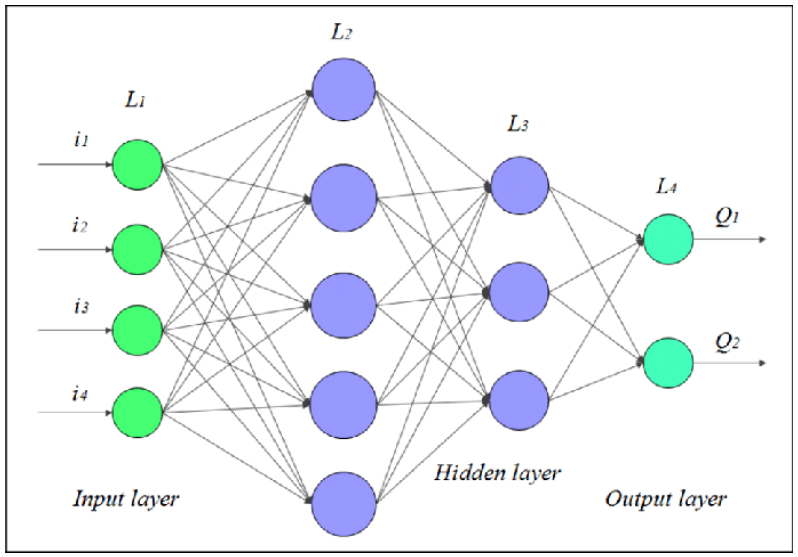




Hasta que no lleguemos a la capa final, calculamos el producto del vector al de la capa actual como matriz de 1 fila por la matriz de pesos de la capa actual. El resultado lo pasamos al vector zl de la capa siguiente. Empezamos desde la capa de entrada

Propagación hacia delante



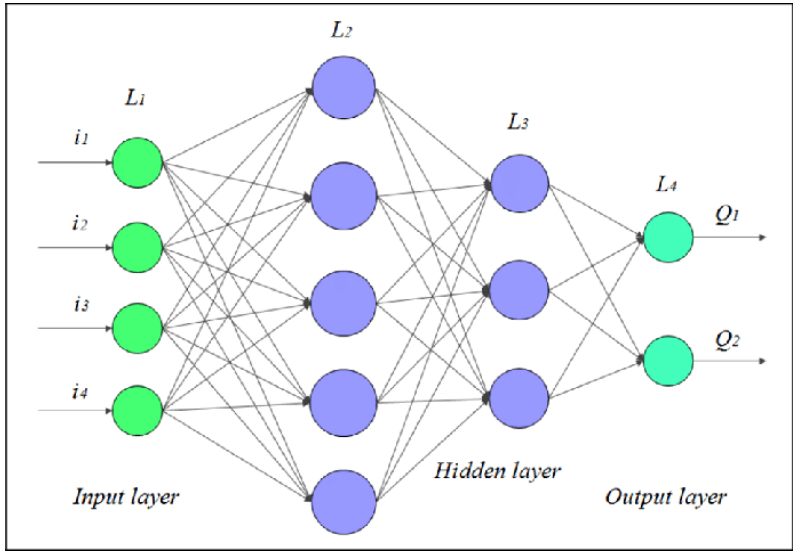


Ahora, sumamos el vector zl de la capa posterior con el vector b de la capa posterior, y copiamos el resultado al vector al de la capa posterior

Propagación hacia delante

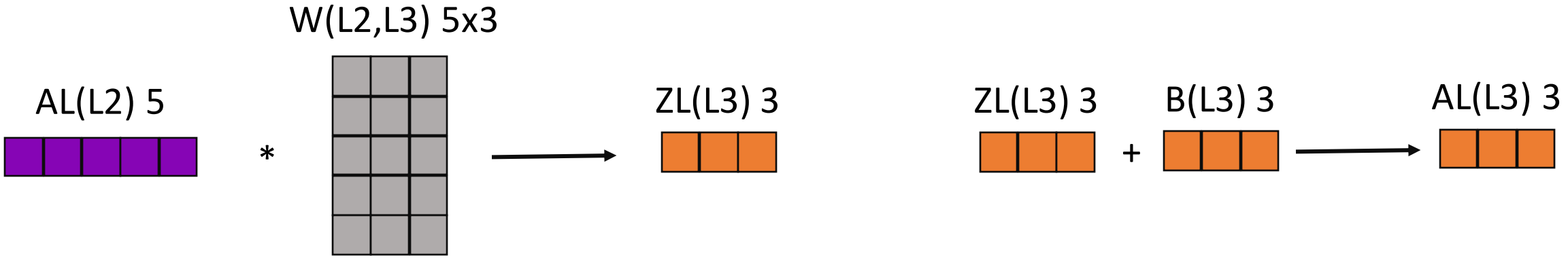
FUNCIÓN_ACTIVACIÓN_CAPA_POSTERIOR (AL(L2) 5)

$sigmoid(al_0)$	$sigmoid(al_1)$	$sigmoid(al_2)$	$sigmoid(al_3)$	$sigmoid(al_4)$

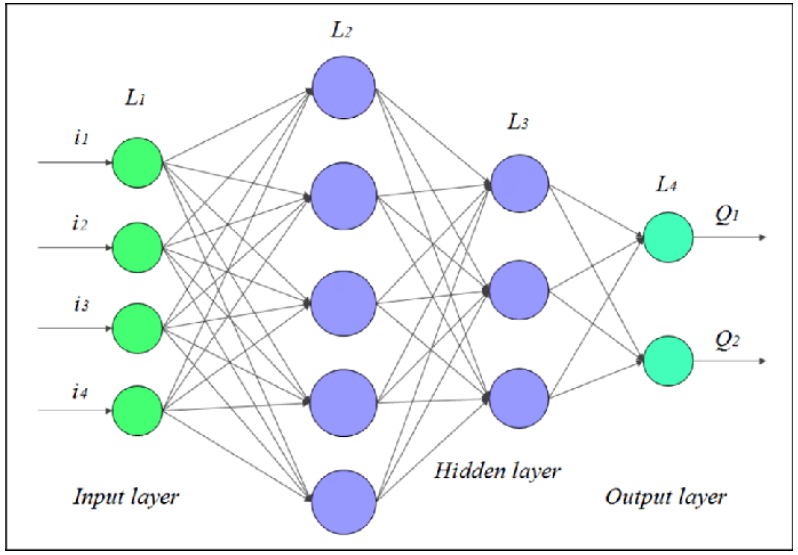


Ahora, aplicamos la función correspondiente a la capa posterior a cada elemento del vector al de la capa posterior. En este ejemplo, consideramos utilizar la función sigmoide para todas las capas, y denotamos cada elemento de al(l2) como a_i, siendo i la posición de cada elemento en el vector.

Propagación hacia delante

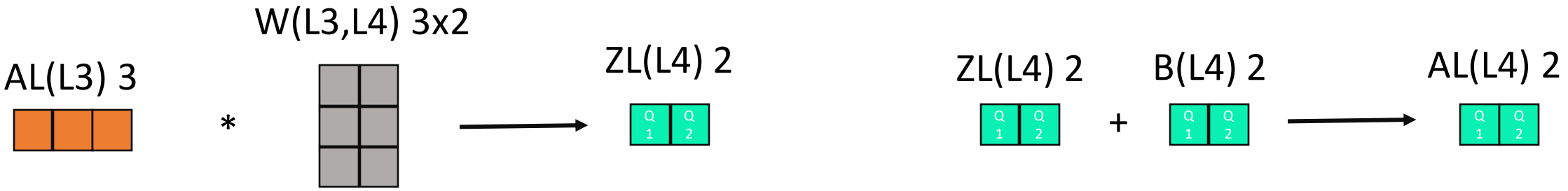


FUNCIÓN_ACTIVACIÓN_CAPA_POSTERIOR (AL(L3) 3)

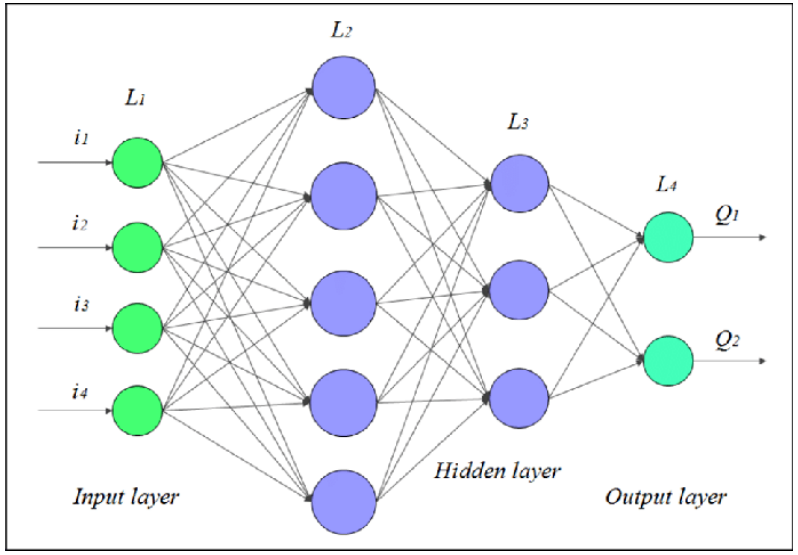


Nos movemos a la siguiente capa y repetimos el proceso

Propagación hacia delante



FUNCIÓN_ACTIVACIÓN_CAPA_SALIDA (AL(L4) 2)



Al final, llegamos a la capa de salida, repetimos el mismo proceso que antes. Al aplicar la función de activación en la capa de salida, el algoritmo finaliza

Nota: en ocasiones, las funciones de activación en la capa de salida pueden depender de los valores del resto de neuronas en la capa de salida, como puede ser el caso de Softmax. En este caso utilizamos la sigmoide, por lo que no es necesario

Propagación hacia delante

Vector_a_matriz_1_fila: recibe como parámetro 1 vector, y devuelve 1 matriz de una fila y tantas columnas como nº elementos del vector
Matriz_1_fila_a_vector: recibe como parámetro 1 matriz de 1 fila, y devuelve un vector equivalente a la fila de dicha matriz
Producto_matrices: recibe como parámetros dos matrices, en orden, m1 y m2, y devuelve otra matriz como el producto m1*m2
Reemplazar_valores_vectores: recibe como parámetros dos vectores, en orden, v1 y v2, copia los valores de v2 en v1
Suma_valores_vectores: recibe como parámetros dos vectores, en orden, v1 y v2, suma los valores de v2 a v1
Aplicar_funcion_vector: recibe como parámetros, en orden, un vector v1 y una función f, y pasa el valor de cada elemento x de v1 a f(x)

Propagación hacia delante

```
sigmoid( x ):
   devolver 1 / ( 1 + e^-x )


propagacion_hacia_delante( vector_valores_entrada, red, funcion_capas_ocultas, funcion_capa_final ):
   Reemplazar_valores_vectores( red.al[0], vector_valores_entrada)
   Índice i desde 0 hasta Red.numero_capas-1 (incluídos):
      Reemplazar_valores_vectores( red.zl[i], Matriz_1_fila_a_vector( Producto_matrices( Vector_a_matriz_1_fila( red.al[i] ), red.pesos[i] ) ) )
      Suma_valores_vectores( red.zl[i], red.bias[i] )
      Reemplazar_valores_vectores( red.al[i], red.zl[i] )
      Si i < Red.numero_capas-2:
         Aplicar_funcion_vector( red.al[i], funcion_capas ocultas )
      Sino:
         Aplicar_funcion_vector( red.al[i], funcion_capa final )
```

Ejemplo de ejecución: propagacion_hacia_delante([1, 0, 0, 0], Red, sigmoid, sigmoid)

Propagación hacia detrás

Acción que permite a la red aprender de los errores que comete al realizar predicciones, modificando los valores de los pesos y biases

Existen algunas diferencias entre algunas redes y otras (a veces no se aplica la función de activación en la capa de salida de la red y las derivadas parciales en dicha capa no la incluyen, en otros casos se calcula el coste como un sumatorio de todos los errores de todos los ejemplos, mientras que en otros casos se calcula el vector gradiente separado para cada uno y luego se suma una media aritmética de dichos errores, en otros casos, se calculan las derivadas parciales del coste con respecto al valor esperado en vez de con respecto el valor predicho, etc.), pero en esencia, comparten las mismas ideas de cómo se calcula el vector gradiente

Aparte, existen diferentes funciones de pérdida para calcular el error cometido por un nodo en la capa de salida, acompañadas de una función de coste para evaluar el error total de la red (media aritmética de todas las funciones de pérdida). Nosotros utilizaremos l**a función de pérdida error cuadrático medio, y la función de coste el error cuadrático medio**.

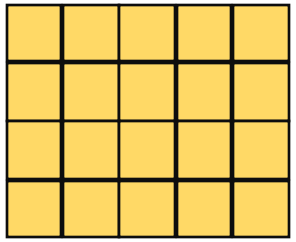
En esta diapositiva, se siguen los pasos de este vídeo: https://www.youtube.com/watch?v=tIeHLnjs5U8&t=3s

Básicamente, la idea es primero calcular un vector que incluye todos los valores que servirán para ajustar los valores de los pesos y los biases de la red, llamado vector gradiente, uno para cada uno de los ejemplos del conjunto de entrenamiento. Después, se sumarán los valores en las mismas posiciones de los vectores gradiente calculados para todos los ejemplos del conjunto de entrenamiento.

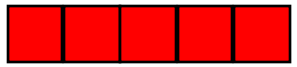
Finalmente, cada valor se dividirá entre el número de ejemplos del conjunto de entrenamiento, y se multiplicará por un hiperparámetro del entrenamiento llamado tasa de aprendizaje, el cual suele ser un número decimal entre 1 y valores cercanos al 0, que permite controlar la velocidad a la que nos aproximamos a un punto dentro de la función de error

Representación v. gradiente matricial:

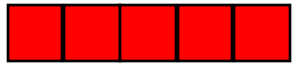
EW(L1,L2) 4x5



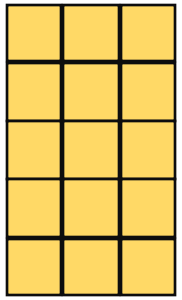
EB(L2) 5



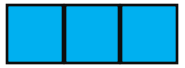
EC(L2) 5



EW(L2,L3) 5x3

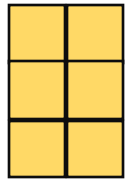


EB(L3) 3

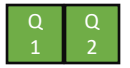


EC(L3) 3

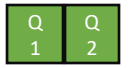
EW(L3,L4) 3x2

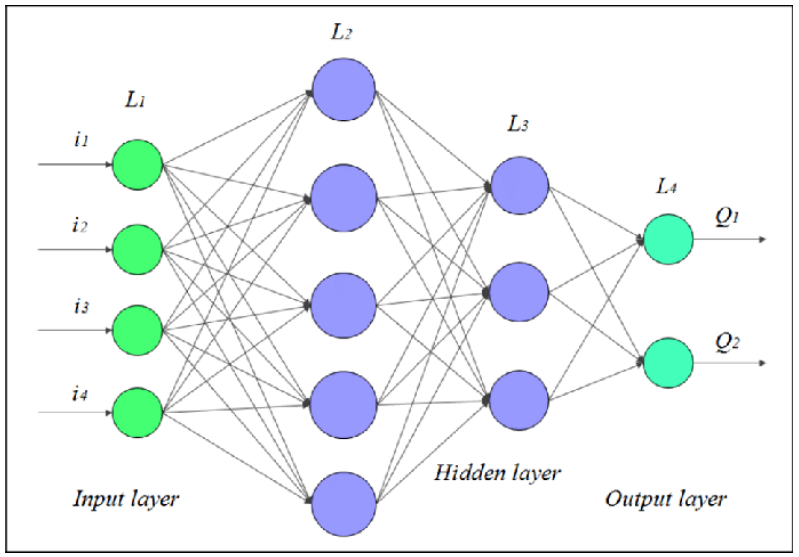


EB(L4) 2



EC(L4) 2





EB: vector donde se guardan los valores de error del bias de la neurona en la capa actual. Sus valores se calcularán al realizar 1 vez la propagación hacia detrás

EC: vector donde se guardan los valores de error de dicha capa con respecto a la función de coste, calculado en la iteración anterior. Sus valores se calcularán al realizar 1 vez la propagación hacia detrás

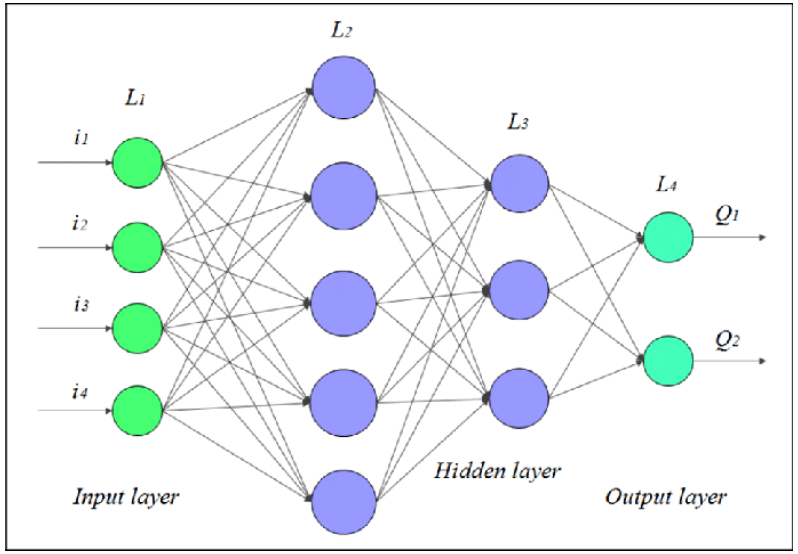
EW: matriz con los valores de los errores de los pesos de las aristas. Sus valores se calcularán al realizar 1 vez la propagación hacia detrás

Nota: no se almacena ni el error del bias ni el error de la capa, para la capa de entrada de la red

Representación v. gradiente como listas

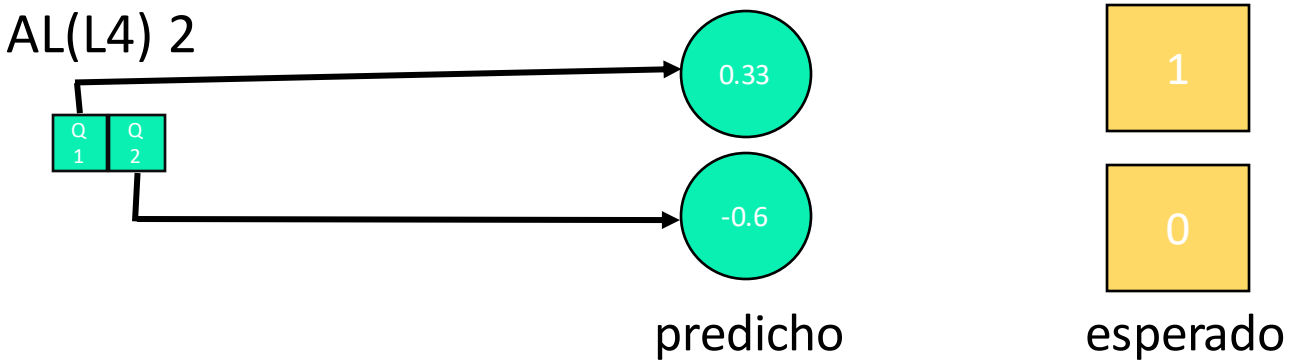
VGrad.err_bias = [
  [vector 5 elementos todos sus valores iguales a 0],
  [vector 3 elementos todos sus valores iguales a 0],
  [vector 2 elementos todos sus valores iguales a 0]
]
VGrad.err_capa = [
  [vector 5 elementos todos sus valores iguales a 0],
  [vector 3 elementos todos sus valores iguales a 0]
]
VGrad.err_pesos = [
  [matriz 4x5 elementos todos sus valores iguales a 0],
  [matriz 5x3 elementos todos sus valores iguales a 0],
  [matriz 3x2 elementos todos sus valores iguales a 0]
]

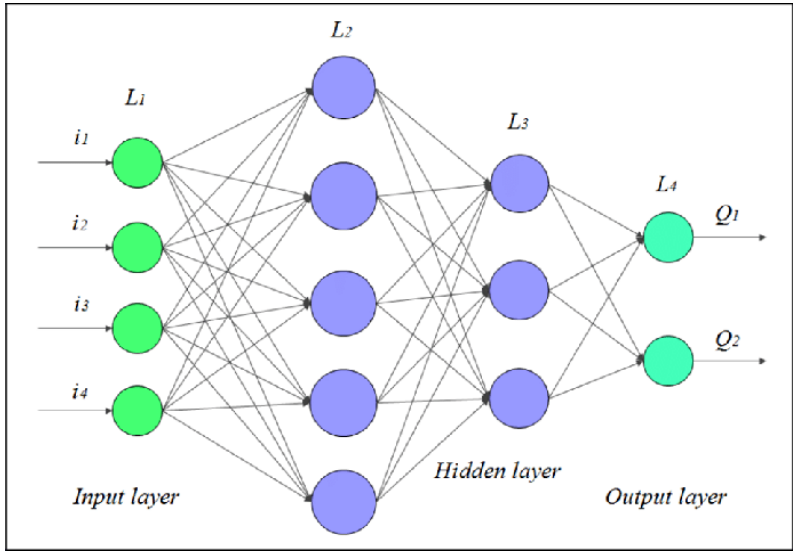
Nota: no se almacena el error del bias para la capa de entrada



Propagación hacia detrás

propagacion_hacia_delante([1, 0, 0, 0], Red, sigmoid, sigmoid)





Empezamos calculando el vector gradiente para un ejemplo del conjunto de entrenamiento. Supongamos que sus valores son [1, 0, 0, 0].

Lo primero que hay que hacer es realizar la propagación hacia delante, para que los valores de los vectores al y zl de la red tengan los valores de la acción de predicción de la red

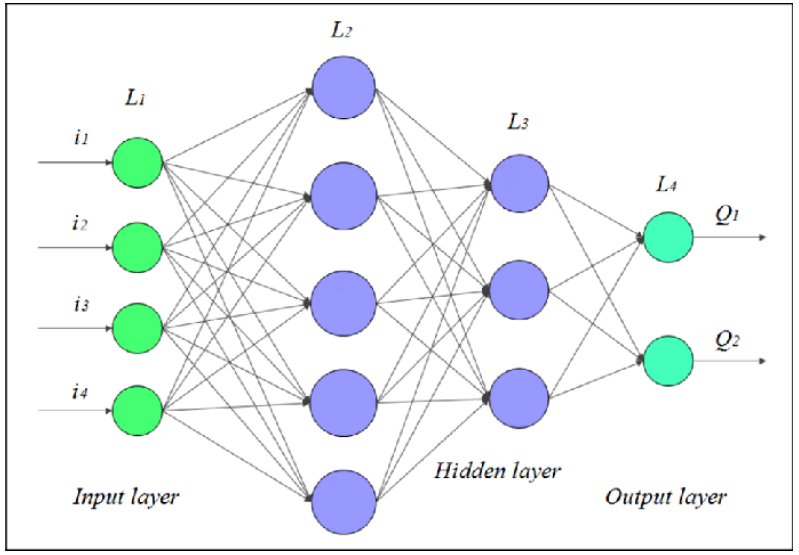
Ahora supongamos que en la capa de salida esperábamos obtener los valores [1, 0], pero nuestra red, sin ajustar, predice los valores [0.33, -0.6]

Propagación hacia detrás

[2* (0.33 - 1) , 2 * (-0.6 - 0)] = [-1,34, -1,2]

[-1,34, -1,2] ⟶ EC(L4) 2 (Q 1, Q 2)

$FC = ((0.33-1)^2 + (-0.6-0)^2)/2 = 0,40445$



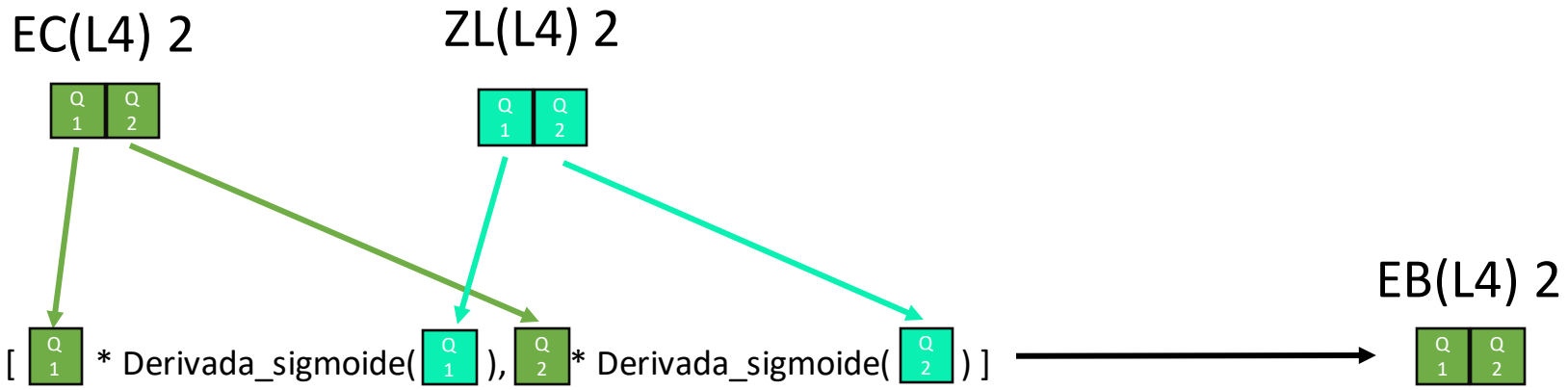
Primero, permaneciendo en la capa de salida, calcularemos el error de los biases de sus neuronas. EL primer paso es guardar en VGrad.err_capa en la capa de salida los valores de esta derivada de la función de pérdida

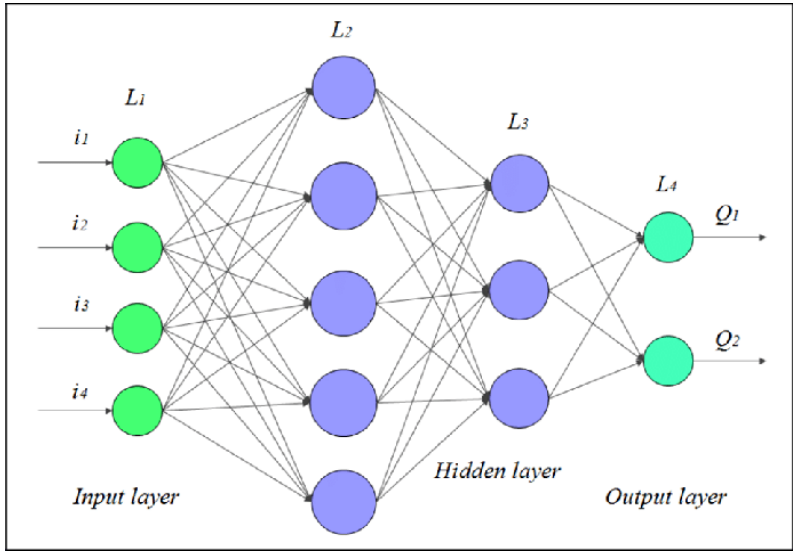
La fórmula de la función de pérdida, recordando que escogimos el error cuadrático, es (x es el valor predicho de una neurona, e y es el valor esperado para dicha neurona): $(x-y)^2$

La derivada de dicha fórmula con respecto a x queda: $2*(x-y) * 1 = 2*(x-y)$

La función de coste (FC) es la media aritmética del sumatorio de $(x-y)^2$ para cada neurona

Propagación hacia detrás



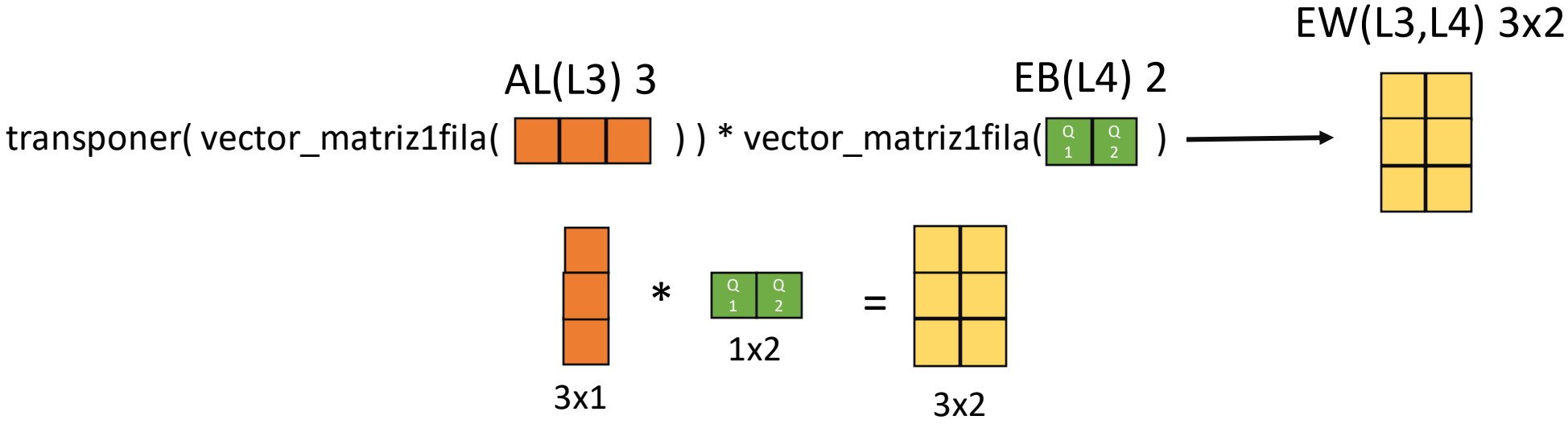


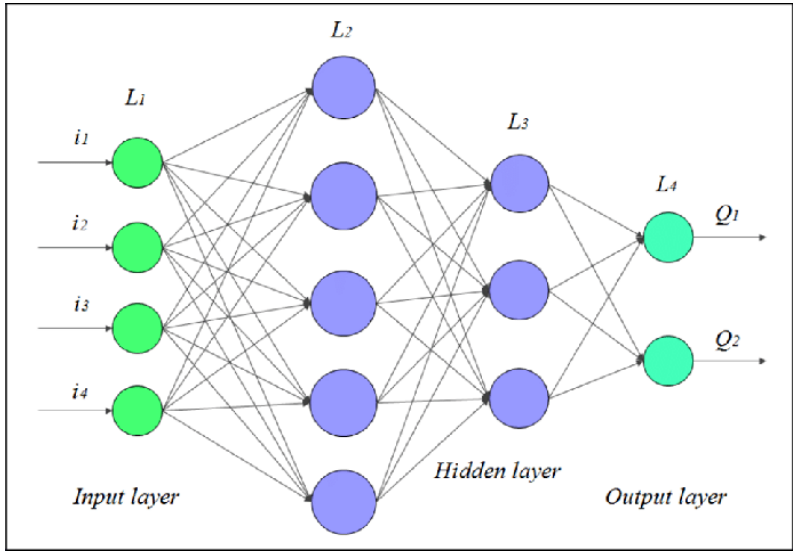
Seguimos en la capa de salida, ahora calcularemos por fin el valor del bias para las neuronas de la capa de salida. Para ello, multiplicaremos cada valor de vector VGrad.err_capa en la capa de salida, calculado anteriormente, por el valor de cada elemento del vector Red.zl en la capa de salida, aplicado a la derivada de la función de activación de esta capa. Pasaremos estos valores al vector VGrad.err_bias de la capa de salida

Podemos expresar la derivada de la función sigmoide como esta función:

```
Derivada_sigmoide(x):
   return sigmoide(x) * (1 - sigmoide(x))
```

Propagación hacia detrás



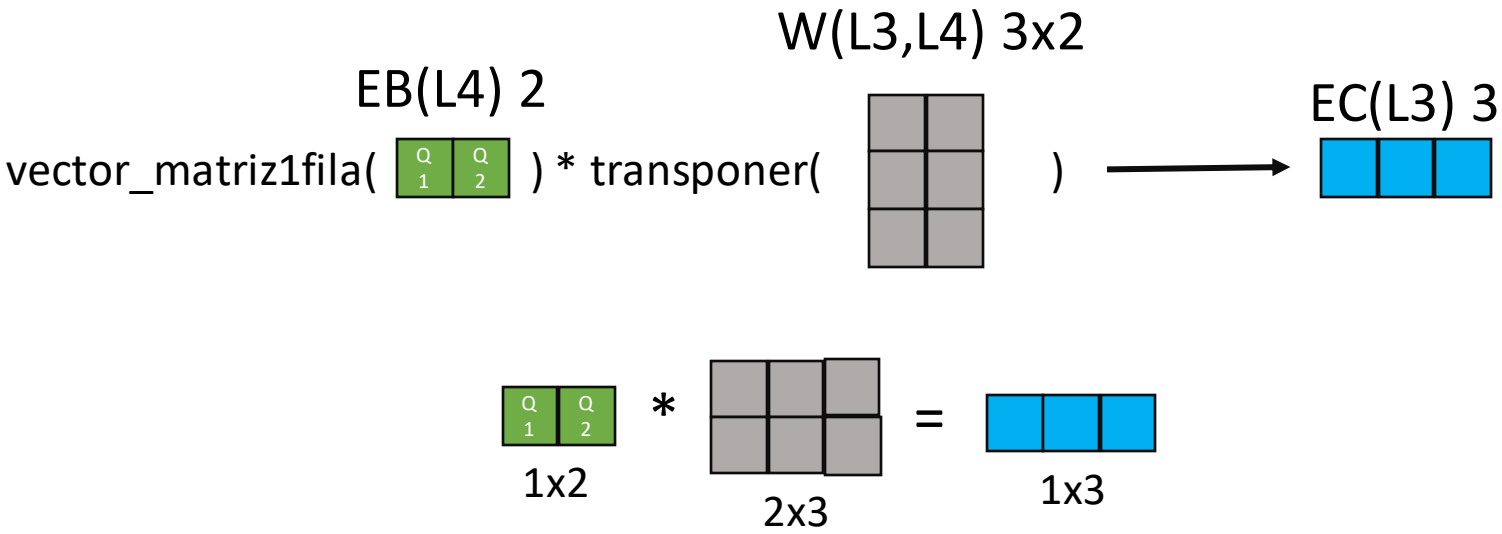


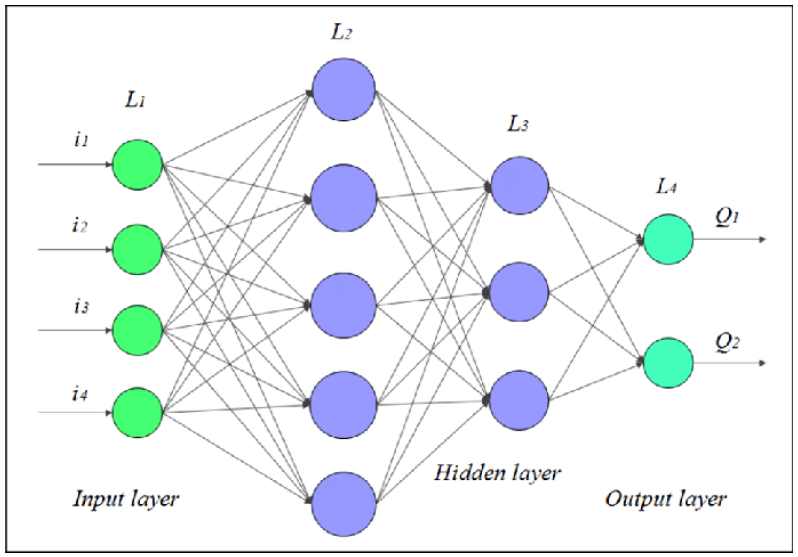
Ahora calcularemos el error de los pesos de las aristas de la capa actual. Para ello, simplemente, calcularemos este producto de matrices. EL resultado lo pasaremos a VGrad.err_pesos en la capa anterior a la capa de salida

La función vector_matriz1fila convierte un vector en una matriz de 1 fila igual al vector

Aparte, la función transponer, transpone la matriz pasada como argumento

Propagación hacia detrás



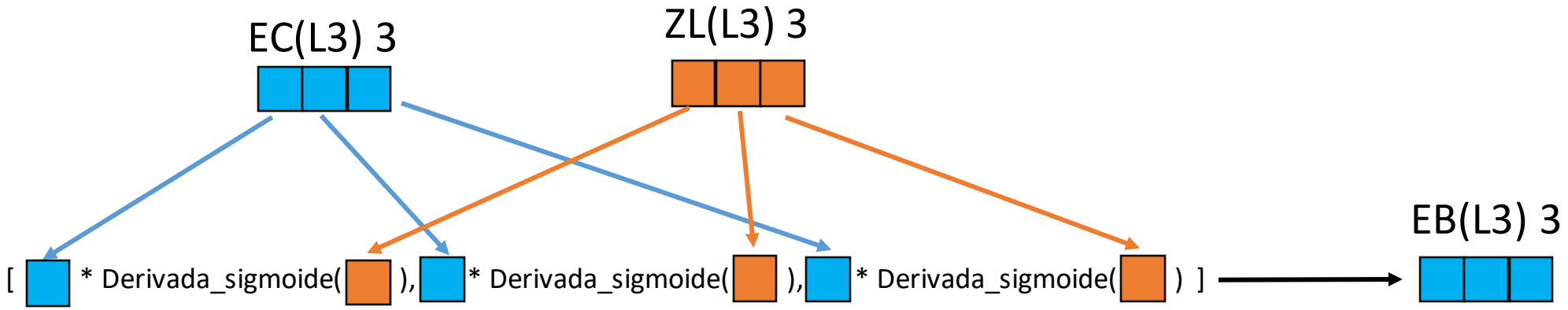


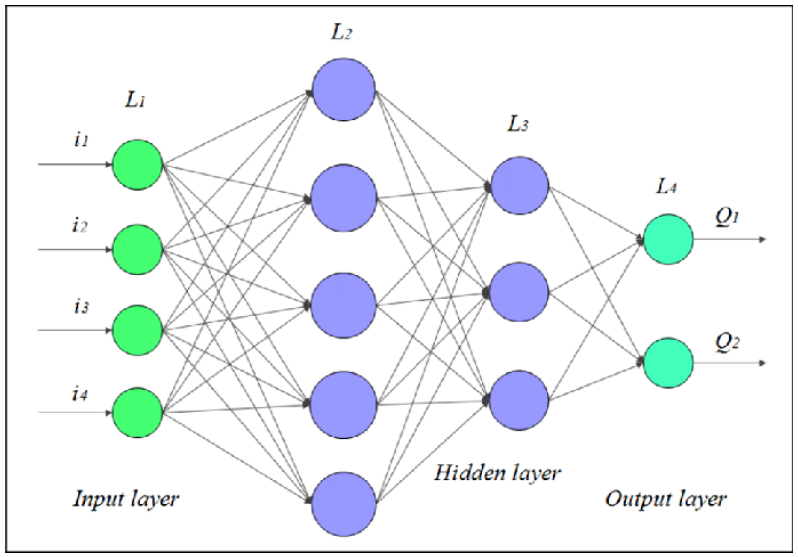
Para terminar en la capa de salida, calcularemos el vector de errores para la capa anterior. EL resultado lo pasaremos a VGrad.err_capa en capa anterior a la capa de salida, convirtiendo la matriz de una fila obtenida como un vector

La función vector_matriz1fila convierte un vector en una matriz de 1 fila igual al vector

Aparte, la función transponer, transpone la matriz pasada como argumento

Propagación hacia detrás

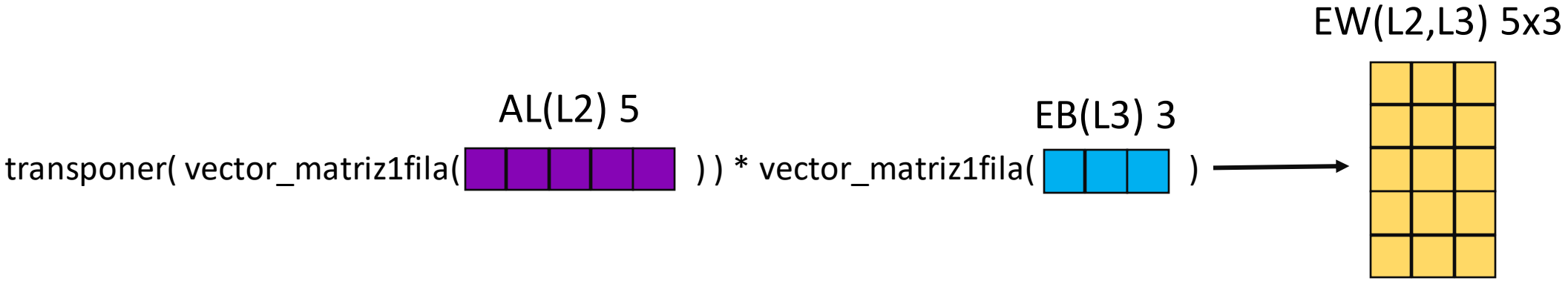


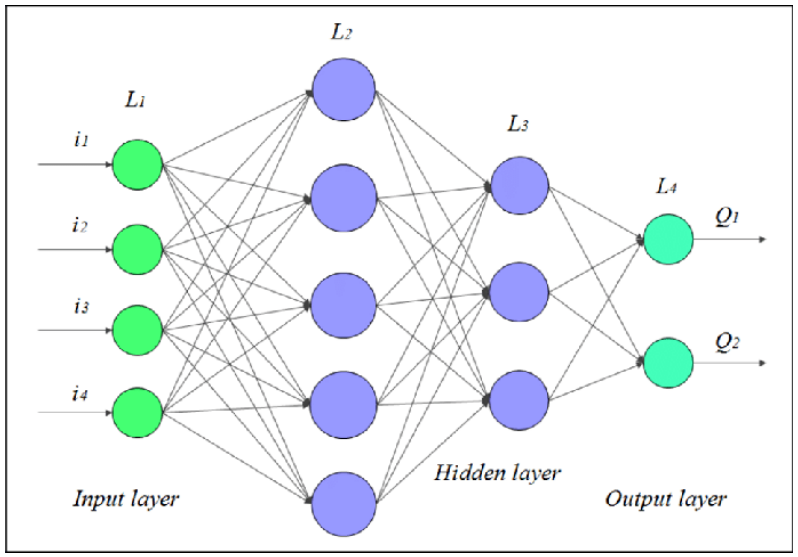


Repetimos los 3 mismos pasos realizados anteriormente, pero ahora con la capa anterior

Empezamos calculando los errores del bias para la que ahora es la capa actual

Propagación hacia detrás

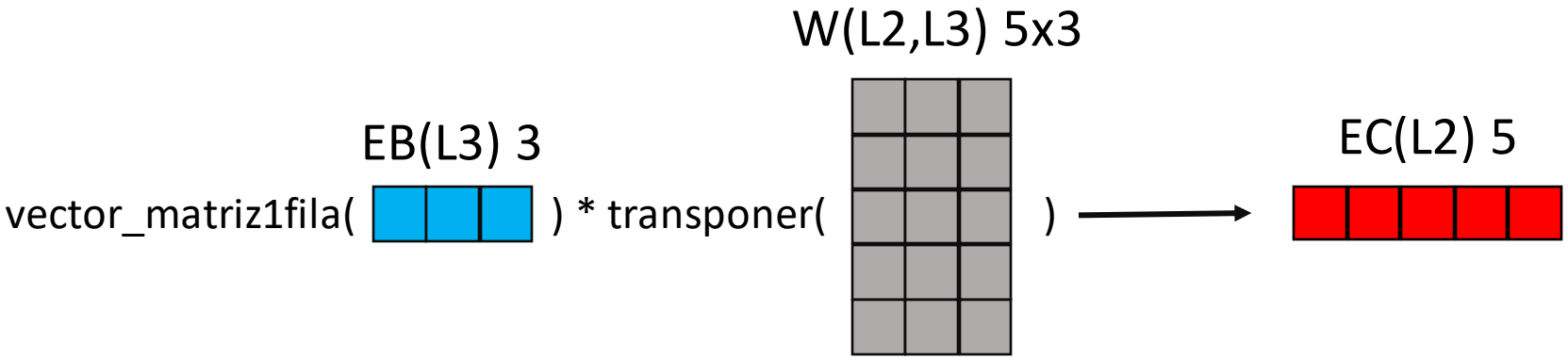


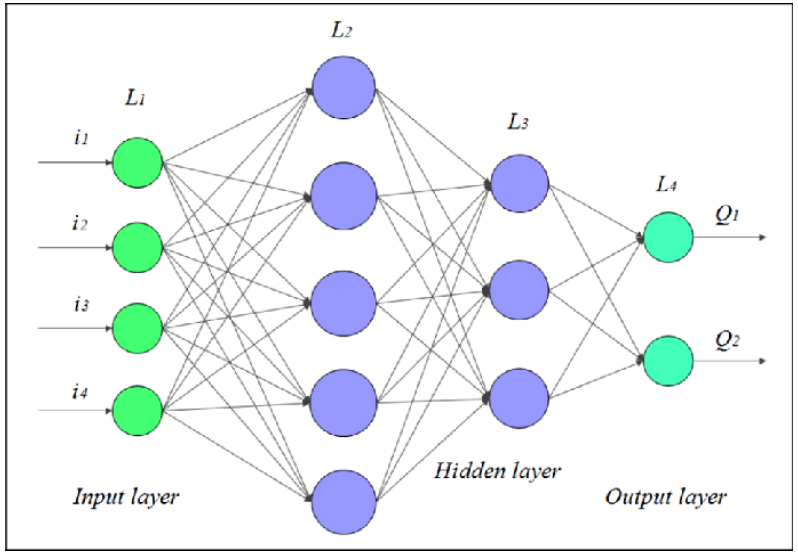


Repetimos los 3 mismos pasos realizados anteriormente, pero ahora con la capa anterior

Después calculamos los errores de los pesos para la que ahora es la capa actual

Propagación hacia detrás

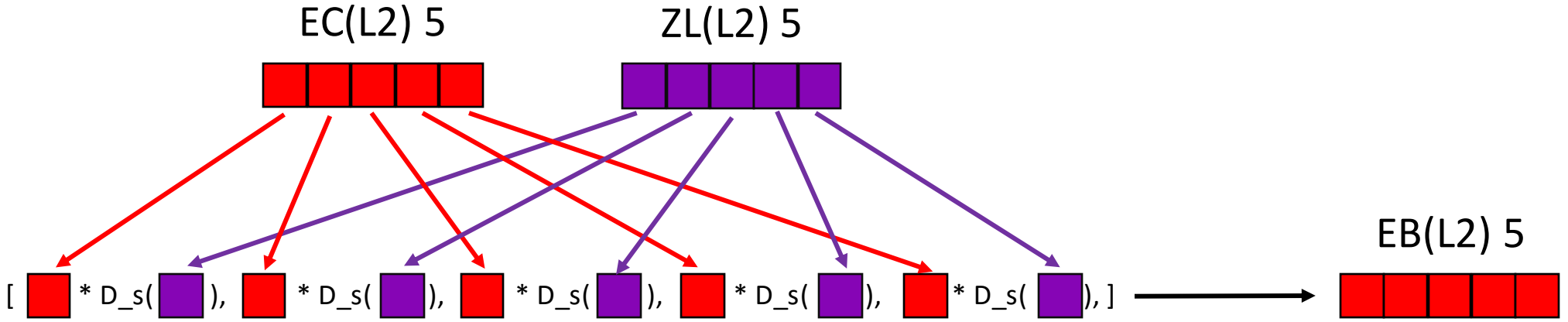


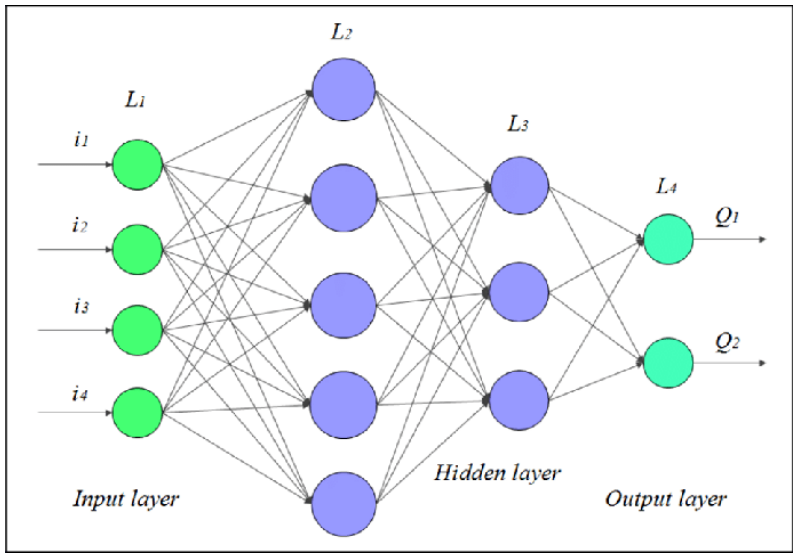


Repetimos los 3 mismos pasos realizados anteriormente, pero ahora con la capa anterior

Para terminar, calculamos el error de la capa anterior con respecto a la que ahora es la capa actual

Propagación hacia detrás



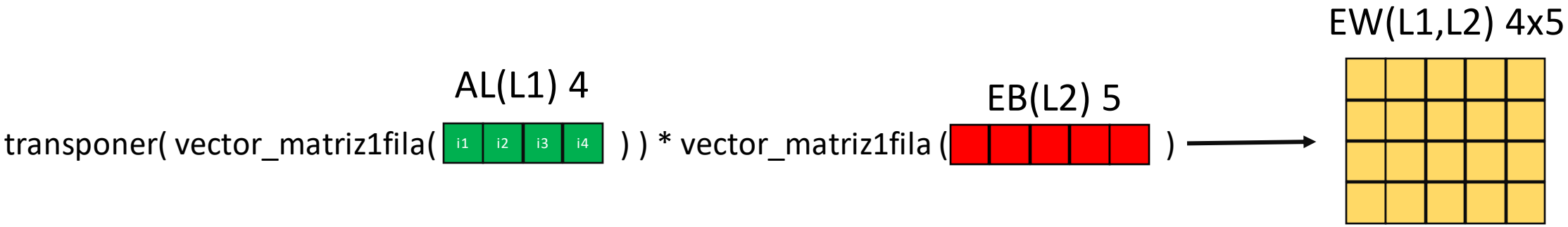


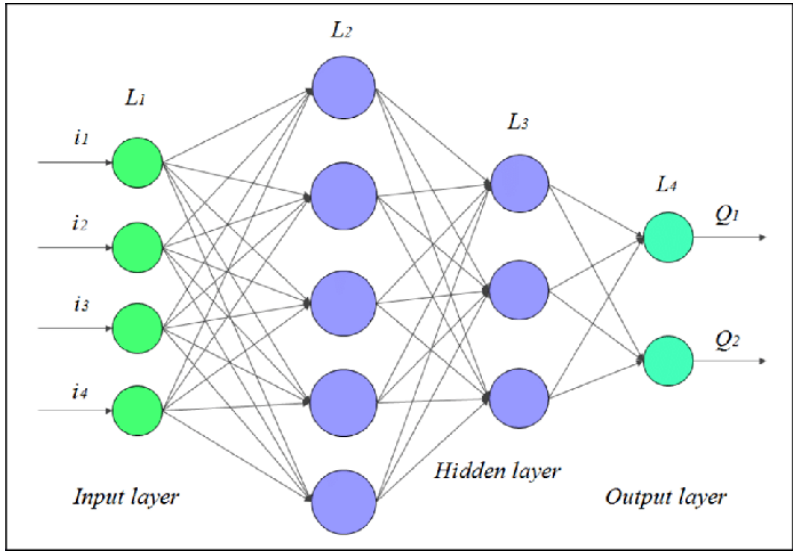
Al retroceder ahora a la capa anterior, en nuestro caso L2, podemos darnos cuenta de que la capa anterior es L1. Recordemos que la capa de entrada no se considera como una capa de neuronas, y es por ello que no hay que calcular ningún error para la capa anterior.

Por ello, solamente nos queda calcular los errores del bias y de los pesos para la capa L2

Aquí calculamos el error del bias. Acortamos Derivada_sigmoide por D_s

Propagación hacia detrás





Al retroceder ahora a la capa anterior, en nuestro caso L2, podemos darnos cuenta de que la capa anterior es L1. Recordemos que la capa de entrada no se considera como una capa de neuronas, y es por ello que no hay que calcular ningún error para la capa anterior.

Por ello, solamente nos queda calcular los errores del bias y de los pesos para la capa L2

Aquí calculamos el error de los pesos

Funciones:

Constructor_vgrad: devuelve una lista con la estructura de la diapositiva 16, pero sin VGrad.error_capa. Recibe una red como parámetro.
Vector_mat1fila: recibe como parámetro 1 vector, y devuelve 1 matriz de una fila y tantas columnas como nº elementos del vector
Mat1fila_vector: recibe como parámetro 1 matriz de 1 fila, y devuelve un vector equivalente a la fila de dicha matriz
Producto_matrices: recibe como parámetros dos matrices, en orden, m1 y m2, y devuelve otra matriz como el producto m1*m2
Transponer_matriz: toma una matriz m como argumento y devuelve su transpuesta, es decir, m^T
Reemplazar_valores_vectores: recibe como parámetros dos vectores, en orden, v1 y v2, copia los valores de v2 en v1
Reemplazar_valores_matrices: recibe como parámetros dos matrices, en orden, m1 y m2, copia los valores de m2 en m1

Se propone una implementación sin necesidad de guardar un vector de error por cada capa

Propagación hacia delante

```
sigmoid( x ):
    devolver 1 / ( 1 + e^-x )
```

```
derivative_sigmoid( x ):
    devolver sigmoid(x) * (1 - sigmoid(x))
```

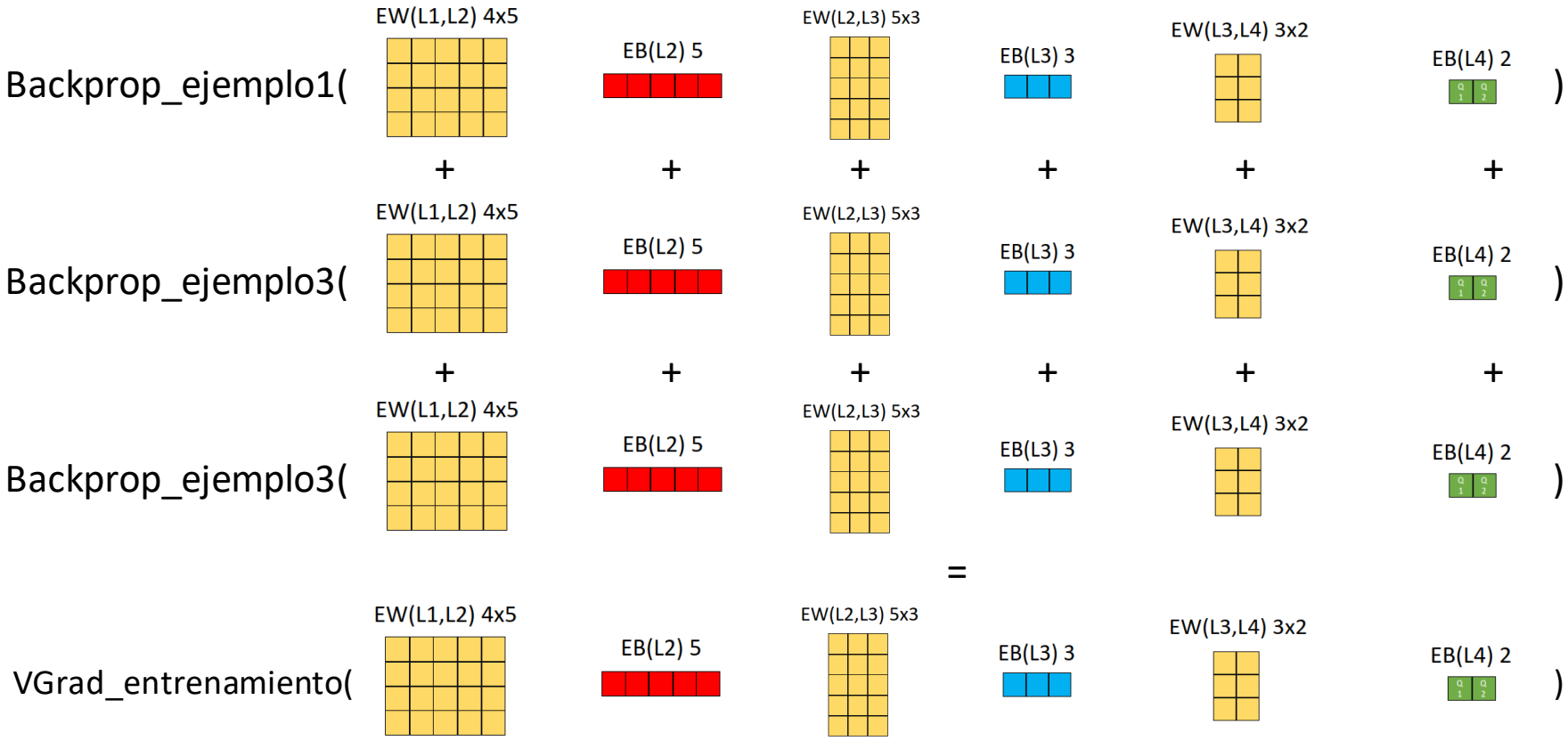
```
vector_gradiente( vector_vals_salida_esp, red, d_funcion_capas_ocultas, d_funcion_capa_salida ):
  vgrad = Constructor_vgrad(Red)
  err_sal = [ 2*( red.al[red.numero_capas-1][i] - vector_vals_salida_esp[i]) para Índice i desde 0 hasta tamaño( red.al[red.numero_capas-1] ) ]
  bcs = [ d_funcion_capa_salida(red.zl[red.numero_capas-2][i])*err_sal[i] para Índice i desde 0 hasta tamaño( err_sal ) ]
  Índice i desde Red.numero_capas-2 hasta 0 (incluídos):
    Reemplazar_valores_vectores( vgrad.err_bias[i], bcs)
    err_pesos = Producto_matrices( Transponer_matriz( Vector_mat1fila( red.al[i] ) ), Vector_mat1fila( bcs ) )
    Reemplazar_valores_matrices( vgrad.err_pesos[i], err_pesos)
    Si i > 0:
      err_capa_ant = Mat1fila_vector( Producto_matrices( Vector_mat1fila( bcs ), Transponer_matriz( red.pesos[i] ) ) )
      bcs = [ d_funcion_capas_ocultas( red.zl[i-1][j] )*err_capa_ant[j] para Índice j desde 0 hasta tamaño(err_capa_ant) ]
  Devolver vgrad
```

Ejemplo de ejecución (primero se debe hacer la propagación hacia delante):

```
propagacion_hacia_delante( [1, 0, 0, 0], Red, sigmoid, sigmoid )

Vector_gradiente_primer_ejemplo = vector_gradiente( [1, 0], Red, derivative_sigmoid, derivative_sigmoid )
```

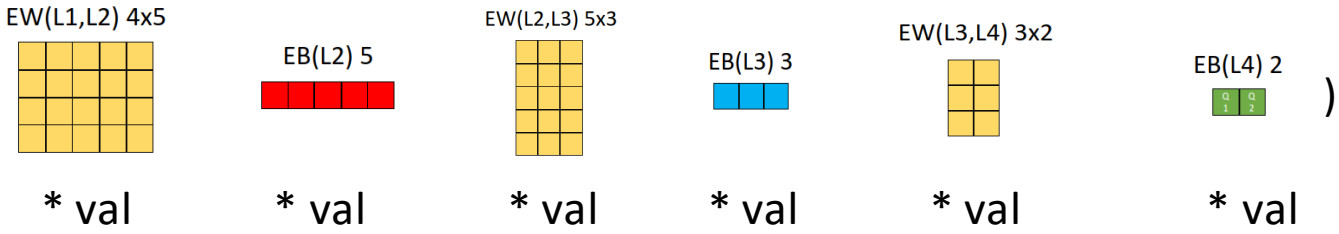
Propagación hacia detrás



Para pasar el vector gradiente a la red para que esta aprenda, teóricamente, primero se van calculando los vectores gradiente para todos los ejemplos del conjunto de entrenamiento con el que queremos entrenar nuestra red, y se suman sus valores. Supongamos que tenemos 3 ejemplos, para cada uno calculamos su vector gradiente, luego sumamos sus valores (para sus matrices de pesos y vectores de biases) y obtenemos un vector gradiente resultado

Propagación hacia detrás

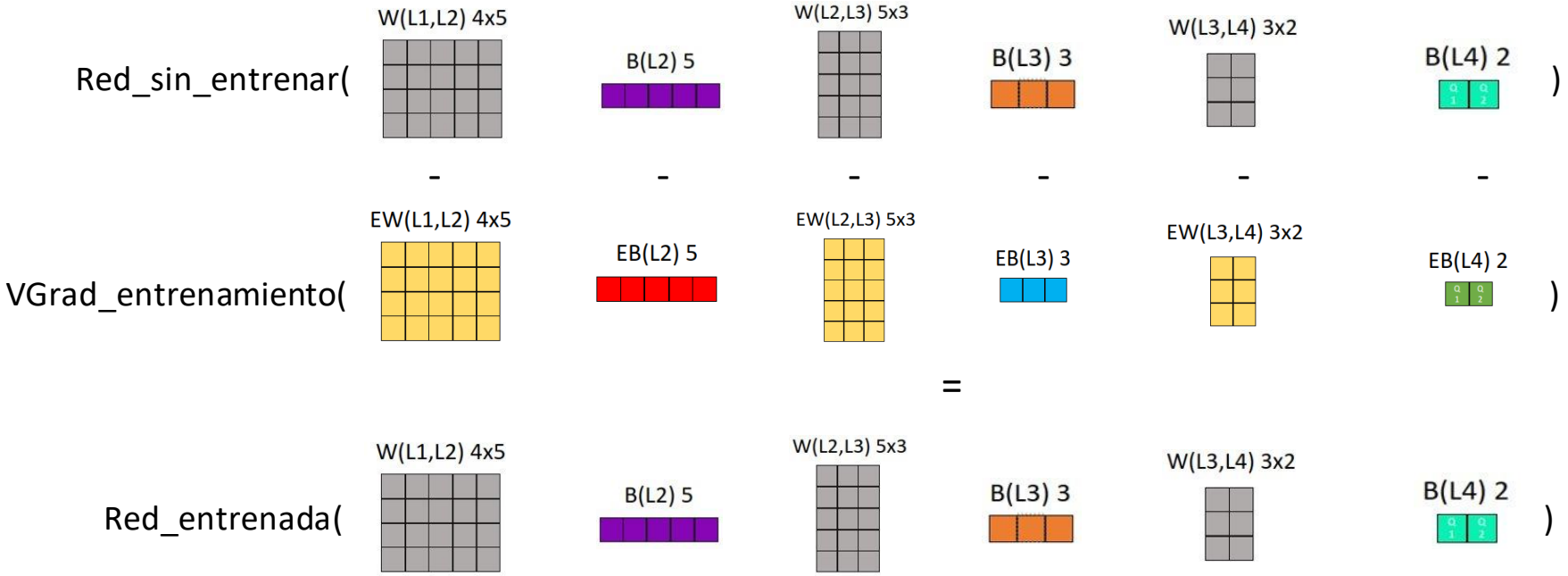
val = tasa_aprendizaje / numero_ejemplos_entrenamiento = "número decimal entre 1 y cercano a 0" / 3

VGrad_entrenamiento(

)

Después hay que multiplicar a cada elemento de cada matriz de pesos y vector de biases, un valor val, el cual es igual a dividir la tasa de aprendizaje escogida entre el número de ejemplos del conjunto de entrenamiento con los cuales se calculó un vector gradiente

Propagación hacia detrás



Finalmente, restamos a la red sin entrenar, para sus correspondientes matrices de pesos y vectores de biases, las matrices de error de pesos y vectores de error de biases del vector gradiente del entrenamiento ajustado del paso anterior. Con ello, ahora nuestra red se habrá ajustado para encajar en nuestro conjunto de datos.

Para que la red aprenda de manera efectiva, todo lo visto hasta ahora se repite cientos, miles o más veces.

Funciones:

Constructor_vgrad: devuelve una lista con la estructura de la diapositiva 16, pero sin VGrad.error_capa. Recibe una red como parámetro.
Suma_valores_vectores: recibe como parámetros dos vectores, en orden, v1 y v2, suma los valores de v2 a v1
Suma_valores_matrices: recibe como parámetros dos matrices, en orden, m1 y m2, suma los valores de m2 a m1
Resta_valores_vectores: recibe como parámetros dos vectores, en orden, v1 y v2, resta los valores de v2 a v1
Resta_valores_matrices: recibe como parámetros dos matrices, en orden, m1 y m2, resta los valores de m2 a m1
Multiplicar_a_vector: recibe como parámetros, en orden, un vector v y un número n, y pasa el valor de cada elemento x de v a n*x
Multiplicar_a_matriz: recibe como parámetros, en orden, una matriz m y un número n, y pasa el valor de cada elemento x de m a n*x

Propagación hacia detrás

```
propagacion_hacia_detrás( conjunto_entrenamiento, red, fun_act, d_fun_act ):
   vgrad = Constructor_vgrad( Red )
   para cada ejemplo en conjunto_entrenamiento:
      propagacion_hacia_delante( ejemplo.valores_entrada, Red, fun_act, fun_act )
      vcalc = vector_gradiente( ejemplo.valores_salida, Red, d_fun_act, d_fun_act )
      Para índice i desde 0 hasta tamaño( vgrad.err_bias ): Suma_valores_vectores( vgrad.err_bias[i], vcalc.err_bias[i] )
      Para índice i desde 0 hasta tamaño( vgrad.err_pesos ): Suma_valores_matrices( vgrad.err_pesos[i], vcalc.err_pesos[i] )
   Para índice i desde 0 hasta tamaño( vgrad.err_bias ):
      Multiplicar_a_vector( vgrad.err_bias[i], tasapren/tamaño(conjunto_entrenamiento) )
      Resta_valores_vectores( red.bias[i], vgrad.err_bias[i] )
   Para índice i desde 0 hasta tamaño( vgrad.err_pesos ):
      Multiplicar_a_matriz( vgrad.err_pesos[i], tasapren/tamaño(conjunto_entrenamiento) )
      Resta_valores_matrices( red.pesos[i], vgrad.err_pesos[i] )
```

Ejemplo de ejecución: propagacion_hacia_detrás(conjunto_entrenamiento, Red, sigmoid, derivative_sigmoid)

En teoría, como se puede ver, se calcula vgrad en función de todos los ejemplos del conjunto de entrenamiento, pero en la práctica, este conjunto de entrenamiento se divide en secciones o baterías (batches en inglés), y se llama a esta función tantas veces como el número de baterías en las que se haya dividido dicho conjunto

Ojo también, aquí hemos considerado utilizar todas las funciones sigmoide, no tiene por qué, por ejemplo, en la capa de salida, se podría haber escogido utilizar Softmax.

Propagación hacia detrás

Cada vez que se hacen todas las operaciones de propagación hacia detrás, hasta ajustar una vez la red al restar el vector gradiente calculado, se está realizando una época de entrenamiento. Se suelen hacer cientos, miles o más épocas para un mismo conjunto de datos, para que la red aprenda mejor, como se mencionó anteriormente.

Como mencionamos, en la práctica no se calcula un vector gradiente de todo el conjunto de entrenamiento, sino que este conjunto se separa en otros subconjuntos llamados baterías (batches en inglés), en cada uno de ellos se ordenan sus elementos de manera aleatoria, y para cada uno de ellos se ejecuta la propagación hacia detrás. Este proceso además se suele repetir varias épocas, hasta conseguir entrenar el modelo para que tenga un mejor rendimiento

Para hacer una aproximación a cómo implementar una red neuronal artificial, se ha mostrado que la tasa de aprendizaje es un hiperparámetro de la red fijo que se escoje para todo el entrenamiento. Sin embargo, esto no es práctico, y se suelen utilizar algunos algoritmos, como ADAM, para ir variando esta tasa de aprendizaje en función de la función de coste y otros criterios, el cual permite llegar a un mínimo local de la función de coste antes (es decir, que permite aprender bien hasta un cierto punto sin desaprender) y además acelerar el proceso de llegada a dicho punto

Aparte, se pueden utilizar diferentes funciones de activación, y sobre todo, para la capa de salida con respecto al resto de capas ocultas, como por ejemplo Softmax, la cual utilizando otra función de coste y de pérdida llamada entropía cruzada, es posible entrenar a las redes neuronales artificiales para resolver mejor problemas de clasificación que una red neuronal como la vista en estas diapositivas (fución de pérdida del error cuadrático y todas las funciones de activación sigmoide)