Redes Neuronales Artificiales Implementación teórica ADAM

Implementación teórica de un algoritmo de optimización de las redes neuronales: ADAM

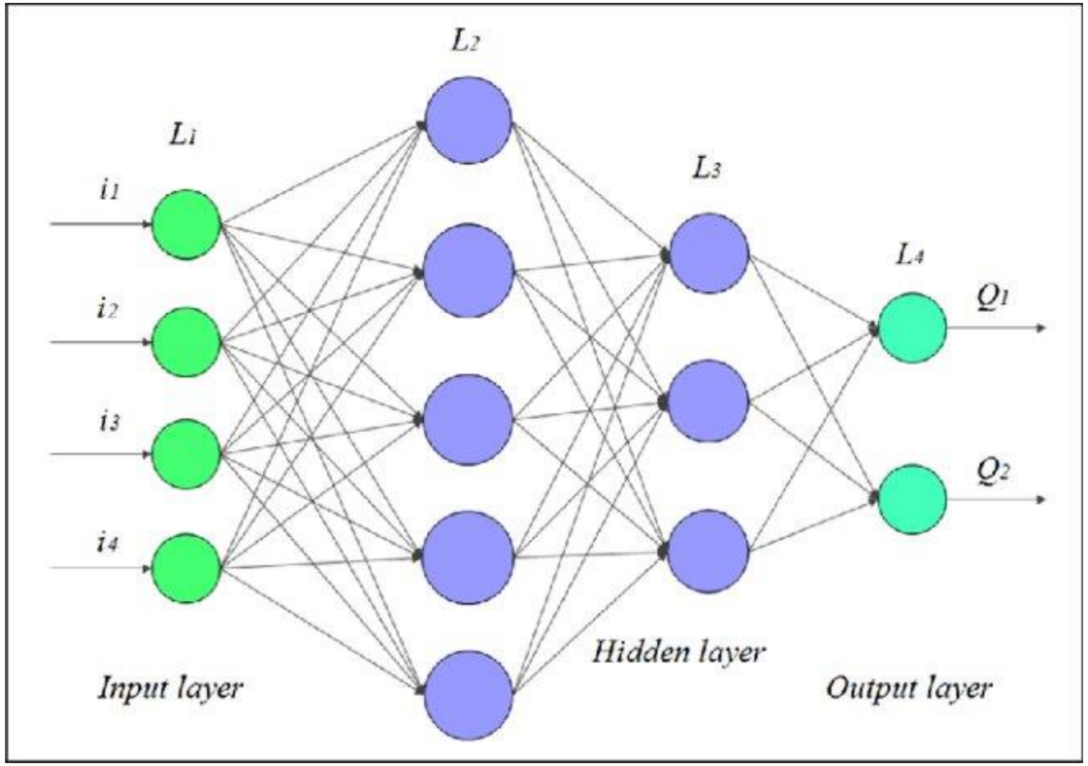
Recordemos

Recordemos que, al entrenar una red neuronal artificial, cambiábamos el valor de un hiperparámetro llamado tasa de aprendizaje, el cual es un número que se multiplicaba a cada valor de error del vector gradiente, ya fuese del error de un peso o el error de un bias.

El problema de restar el vector gradiente solamente multiplicando la tasa de aprendizaje es que la velocidad de aprendizaje se mantiene constante, por lo que no se puede llegar a encontrar antes un mínimo local

ADAM es un algoritmo de optimización que permite acercarse antes a un mínimo local, con menos iteraciones. Tiene una serie de desventajas, como por ejemplo encontrar mínimos locales muy pronunciados, que no suelen ser los mejores, o quedarse en un mínimo local y de ahí no moverse. Se puede minimizar el impacto de estos problemas añadiendo más capas a la red, aumentar el número de neuronas en cada red (sin pasarse), normalizar los datos del conjunto de entrenamiento (a veces se hace con respecto a todo el conjunto, otras con respecto a un subconjunto del conjunto del entrenamiento al hacer SGD, etc...)

Nos estamos basando



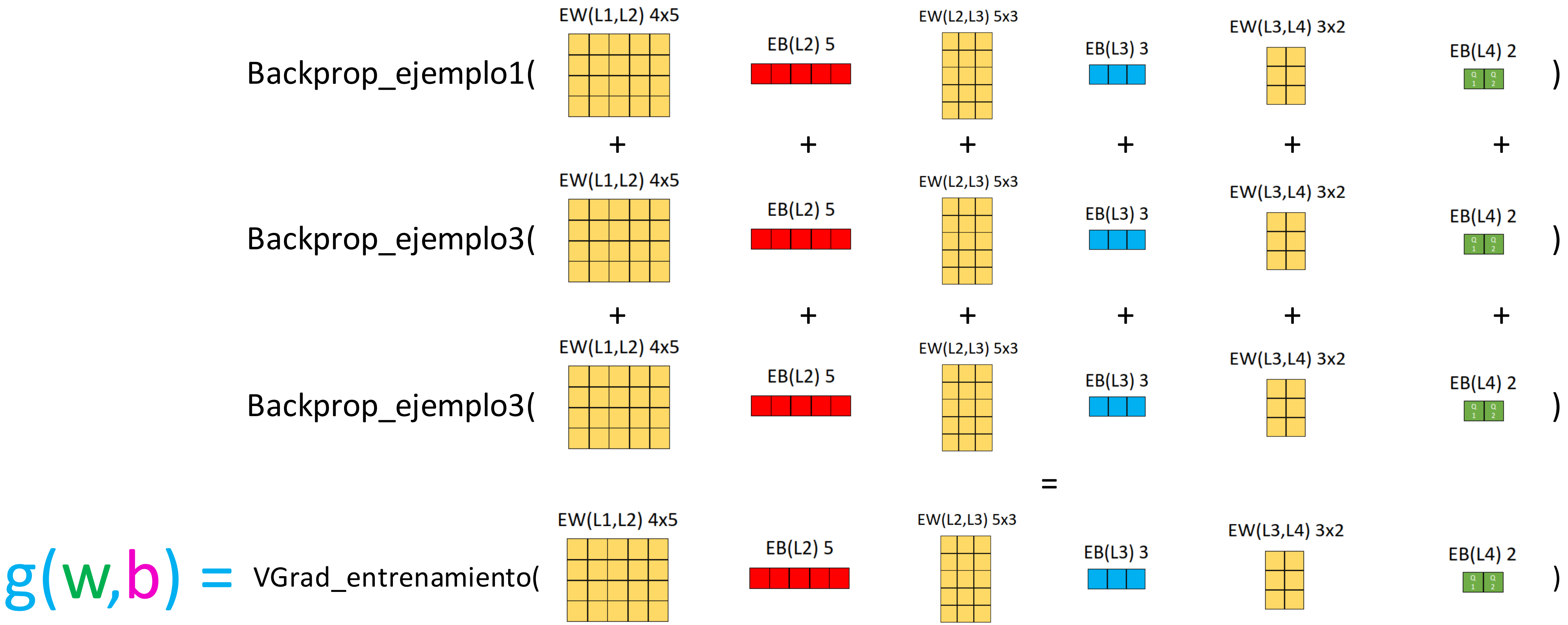
En la misma red neuronal artificial vista en las anteriores diapositivas. Solamente recordemos la representación en forma matricial de su vector gradiente, que se verá a continuación

Hiperparámetros

Este algoritmo necesita de unos cuantos hiperparámetros, uno de ellos ya visto:

- Tasa_aprendizaje: la tasa de aprendizaje ya vista en anteriores presentaciones
- B1: valor decimal dentro del rango [0,1) (el 1 sin incluir)
- B2: valor decimal dentro del rango [0,1) (el 1 sin incluir)
- Epsilon: valor muy cercano al 0, su valor suele ser 10^{-8}

Algoritmo

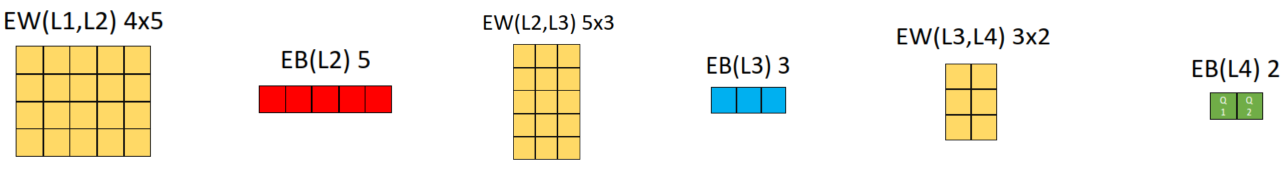


Supongamos que disponemos del vector gradiente del entrenamiento (VGrad_entrenamiento visto anteriormente, que era la suma de los vectores de bias y las matrices de pesos de los vectores gradiente calculados para cada ejemplo del conjunto de entrenamiento)

Algoritmo

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

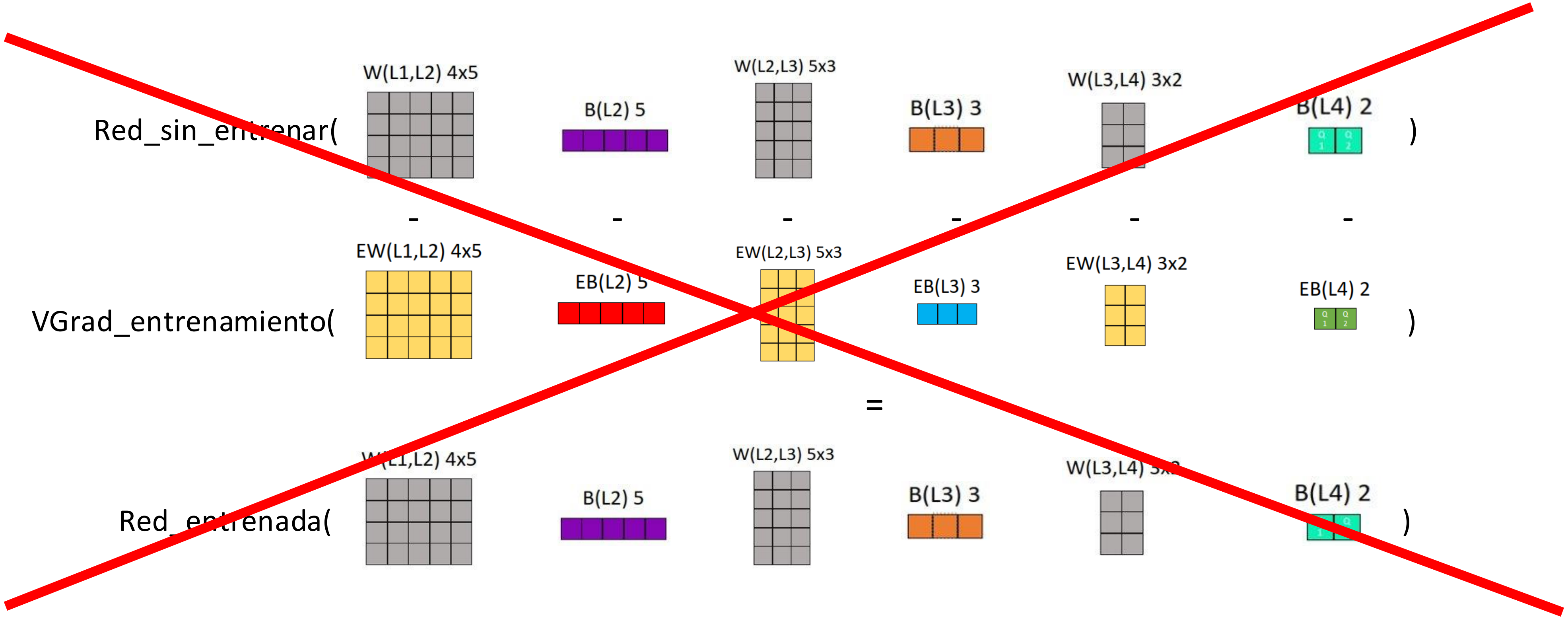
g(w,b) = VGrad_entrenamiento(



* val * val * val * val * val * val)

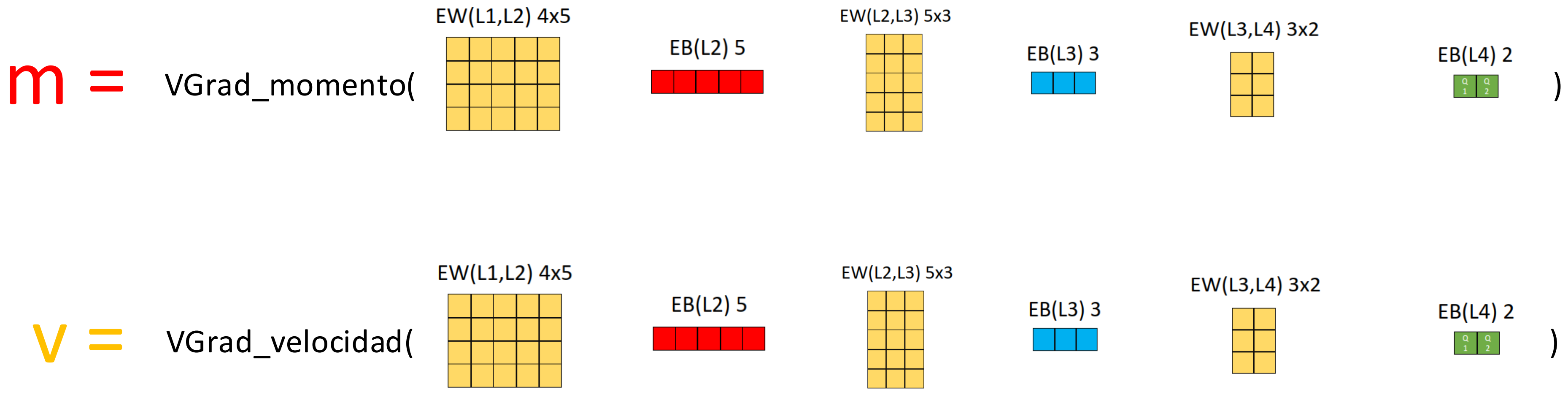
Lo siguiente que se tendrá que hacer es dividir cada valor de error entre el número de ejemplos del conjunto de entrenamiento utilizados, que recordemos en este ejemplo, eran 3. No se multiplica la tasa de aprendizaje, ya que esta se aplicará más adelante en una fórmula en concreto

Algoritmo



Ahora, no restaremos este vector gradiente a la red sin aprender, como vimos en anteriores presentaciones, sino que calcularemos, como hemos dicho, para cada valor de error de bias o de peso dos valores, uno llamado momento, y otro velocidad, que se utilizarán en una fórmula que se verá más adelante

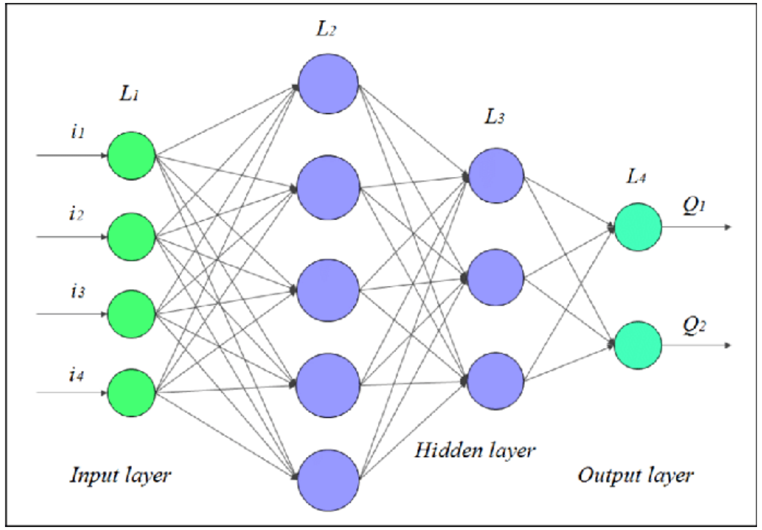
Algoritmo

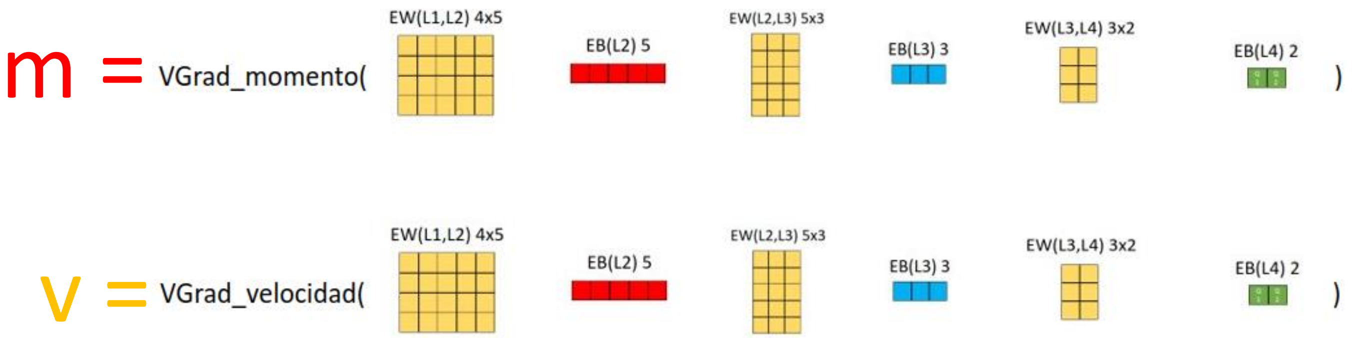


Para empezar, crearemos dos "vectores gradiente" (entre comillas porque tienen la misma estructura , pero no son lo mismo), **con todos los valores de las matrices de errores de peso, y los vectores de erorres de biases, inicializados a 0**

Representación v. ADAM como listas

VAdam.momento_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]
]
VAdam.momento_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]
]
VAdam.velocidad_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]
]
VAdam.velocidad_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]
]





Algoritmo

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1)\mathbf{g}(\mathbf{w}_t)$$

$$\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2)\mathbf{g}(\mathbf{w}_t)^2$$

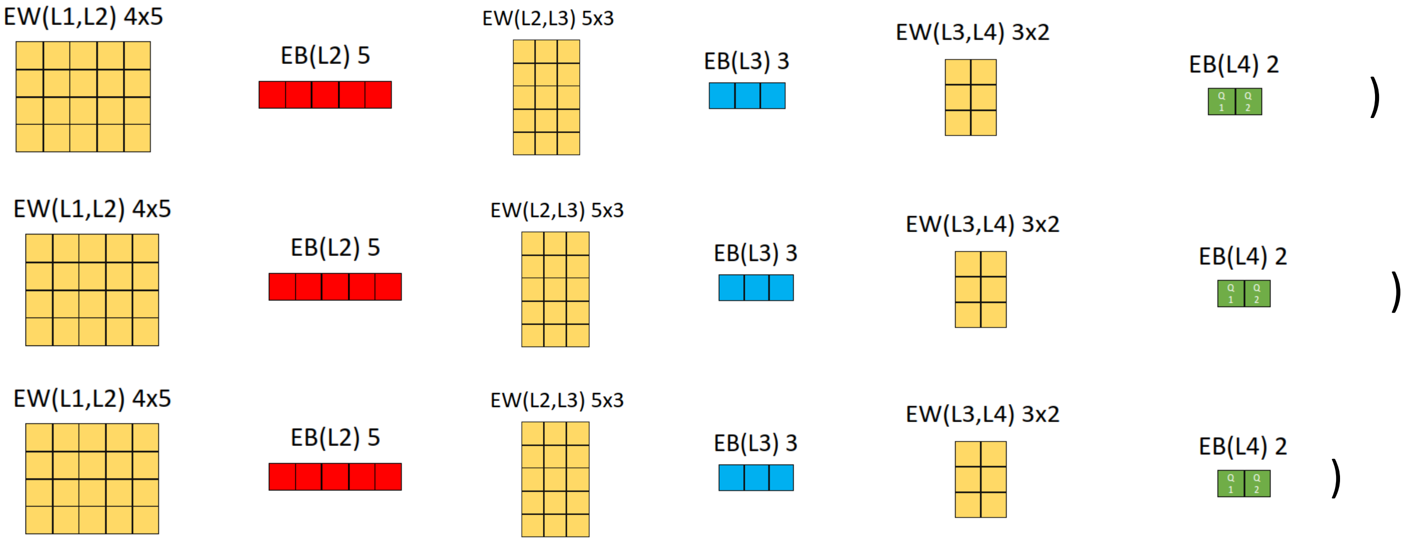
$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1)\mathbf{g}(\mathrm{b}_t)$$

$$\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2)\mathbf{g}(\mathrm{b}_t)^2$$

m = VGrad_momento( )

v = VGrad_velocidad( )

g(w,b) = VGrad_entrenamiento( )



Recordemos, este algoritmo se aplica una vez hemos obtenido el vector gradiente fruto del entrenamiento. Con él vamos a modificar el valor de cada elemento en los "gradientes" de momento y velocidad en base a estas fórmulas. Se aplica lo mismo para cada peso (w en verde) como para cada bias (b en rosa). g(w) en azul es el valor de dicho peso en el vector gradiente, y g(b) en azul es el valor de dicho bias en el vector gradiente. Recordar los hiperparámetros B1 y B2 mencionados anteriormente

Recordemos, todos los valores en momento y velocidad son 0, por tanto, para t=0, mt = 0 y vt = 0

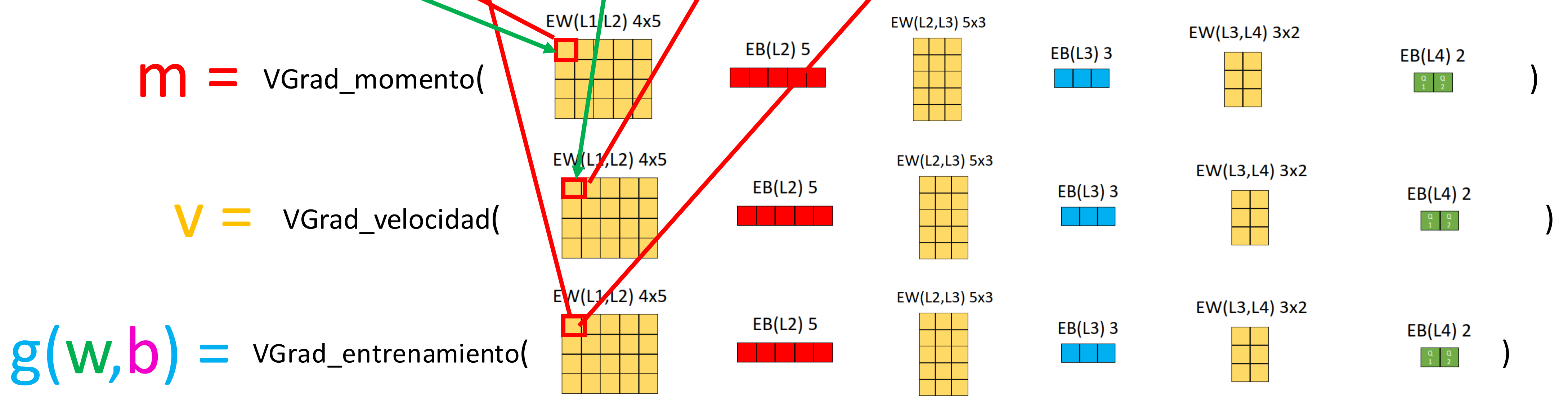
Algoritmo

m_{t+1}= B1*m_t + (1-B1)*g(w_t)

v_{t+1}= B2*v_t + (1-B2)*g(w_t)

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1)\mathbf{g}(\mathbf{w}_t)$$

$$\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2)\mathbf{g}(\mathbf{w}_t)^2$$



Ejemplo para uno de los pesos. La flecha roja indica que primero esos valores se leen para calcular el resultado, y la flecha verde indica dónde se almacena el resultado una vez calculado. Como se puede observar, se considera para cada cálculo la misma posición en la misma matriz para los tres vectores, el de momento, el de velocidad y el del gradiente

Se deben hacer las mismas operaciones para todos los elementos de las matrices EW(L1,L2), EW(L,2L3) y EW(L3,L4), tanto para en el vector momento como para el vector velocidad

Algoritmo

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1-\beta_1)\mathbf{g}(\mathrm{b}_t)$$

$$\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1-\beta_2)\mathbf{g}(\mathrm{b}_t)^2$$

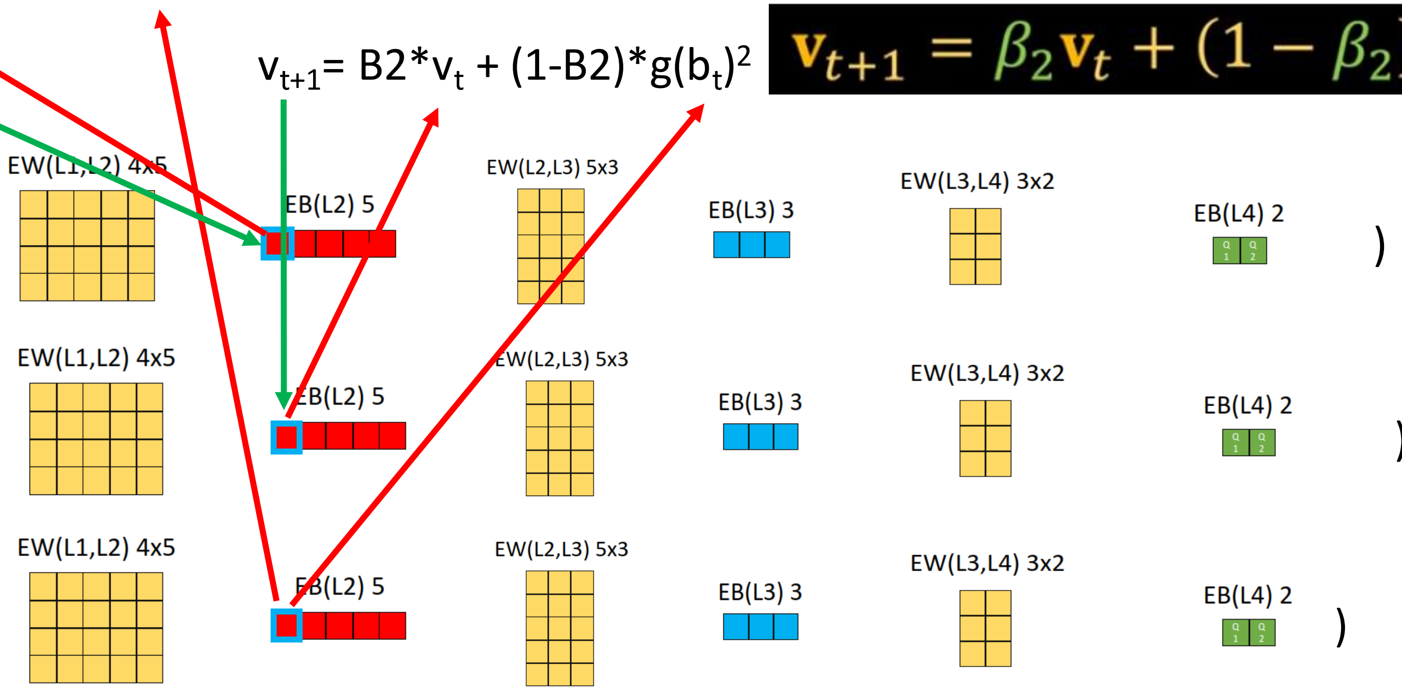
$m_{t+1}= B1*m_t + (1-B1)*g(b_t)$

$v_{t+1}= B2*v_t + (1-B2)*g(b_t)^2$

m = VGrad_momento( )

v = VGrad_velocidad( )

g(w,b) = VGrad_entrenamiento( )



Se hace el mismo cálculo para los vectores con los biases de la red. Los cálculos deben hacerse en todos los elementos de EB(L2), EB(L3) y EB(L4), tanto para el vector momento como para el vector velocidad

Funciones:

Constructor_vadam: devuelve una lista con la estructura de la diapositiva 9

```
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], 1/tamaño(conjunto_entrenamiento) )
   Para índice i desde 0 hasta tamaño( vgrad.err_pesos ):
      Multiplicar_a_matriz( vgrad.err_pesos[i], 1/tamaño(conjunto_entrenamiento) )
   Devolver vgrad
```

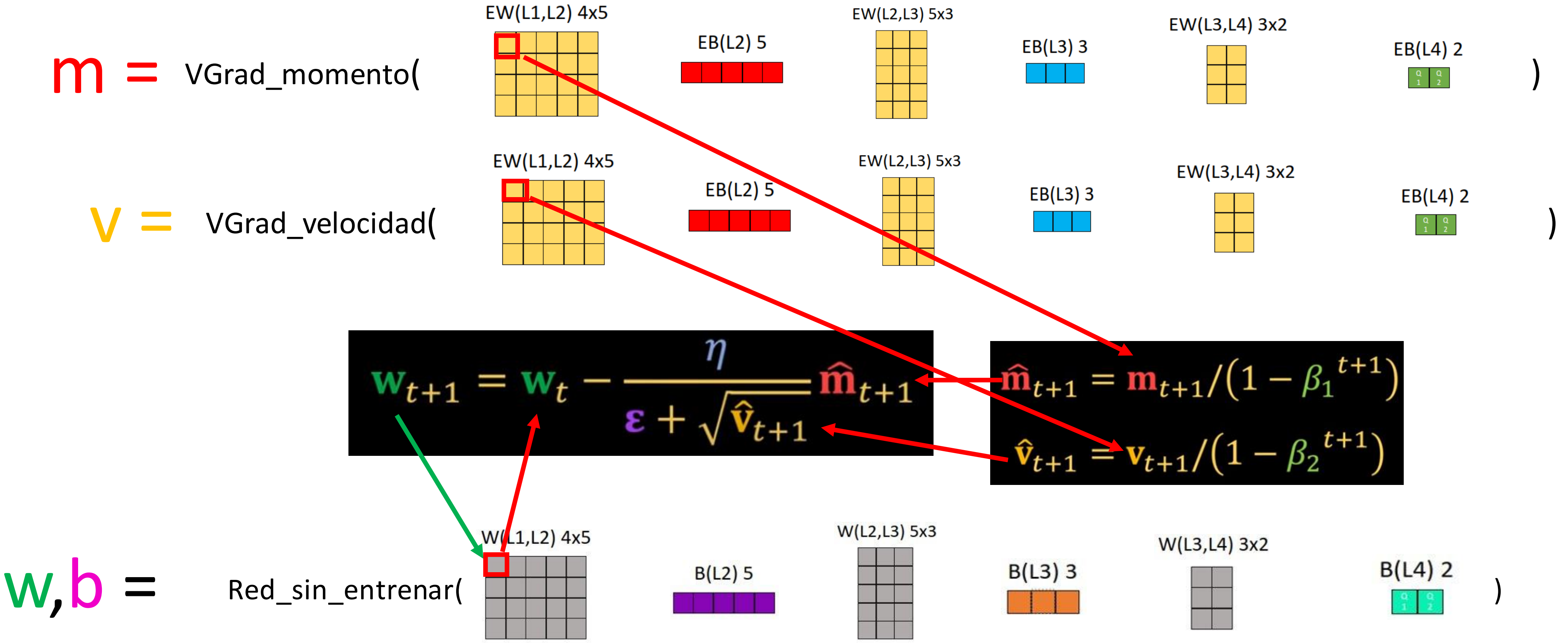
Algoritmo

```
def updateAdamVectorValues(  red, vadam, vgrad, b1, b2 ):
    Para índice l desde 0 hasta red.numero_capas-1:  (incluídos):
        Para índice f desde 0 hasta numero_filas(  vgrad.error_pesos[l]  ): (incluídos):
            Para índice c desde 0 hasta numero_columnas(  vgrad.error_pesos[l]  ): (incluídos):
                vadam.momento_pesos[l][f][c]  = b1 * vadam.momento_pesos[l][f][c]  + (1 - b1) * vgrad.error_pesos[l][f][c]
                vadam.velocidad_pesos[l][f][c]  = b2 * vadam.velocidad_pesos[l][f][c]  + (1 - b2) * (vgrad.error_pesos[l][f][c])^2
        Para índice i desde 0 hasta tamaño(  vgrad.error_bias[l]  ): (incluídos):
            vadam.momento_bias[l][e]  = b1 * vadam.momento_bias[l][e]  + (1 - b1) * vgrad.error_bias[l][e]
            vadam.velocidad_bias[l][e]  = b2 * vadam.velocidad_bias[l][e]  + (1 - b2) * (vgrad.error_bias[l][e])^2
```

Ejemplo de ejecución:

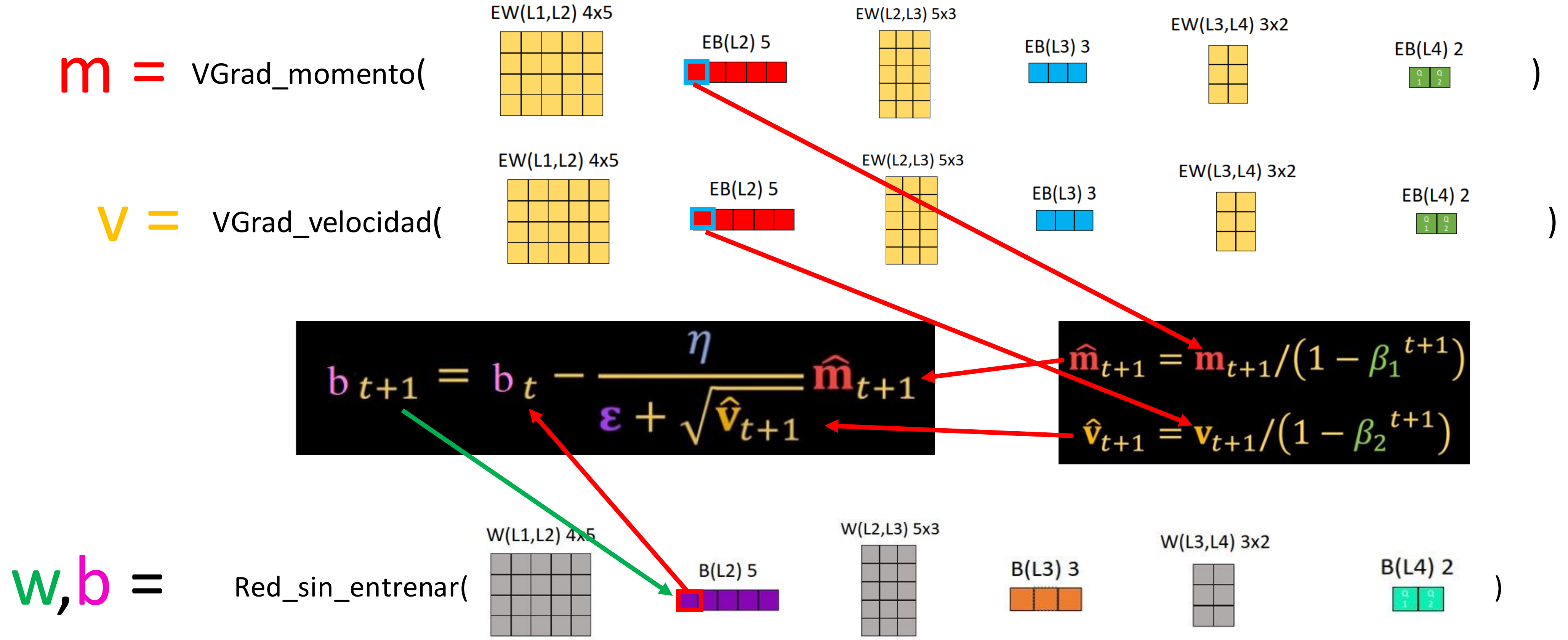
```
VAdam = Constructor_vadam(  Red )
Para época en épocas:
    VGrad = propagación_hacia_detrás(  conjunto_entrenamiento,  Red, sigmoid, derivative_sigmoid  )
    updateAdamVectorValues(  Red, VAdam, VGrad, 0.9, 0.99 )
```

Algoritmo



Ahora, tendremos que restar cada elemento en cada matriz de pesos de la red, un valor que se calcula para cada mismo elemento en los vectores momento y velocidad previamente calculados. Las fórmulas son las que se muestran en las imágenes. Aparte, identificamos η como la tasa de aprendizaje, y ε como el hiperarámetro epsilon, que era un valor cercano a 0. Aparte, observemos que se ajusta tanto m_{t+1} como v_{t+1} antes de restar el resultado al correspondiente peso en la red. Se hace esta operación para todos los elementos de las matrices W(L1,L2), W(L2,L3) y W(L3,L4).

Algoritmo



Haremos lo mismo para los biases de la red. De la misma manera, se repiten las operaciones para los vectores de biases B(L2), B(L3) y B(L4)

Funciones:

Constructor_vadam: devuelve una lista con la estructura de la diapositiva 9

```
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], 1/tamaño(conjunto_entrenamiento) )
   Para índice i desde 0 hasta tamaño( vgrad.err_pesos ):
      Multiplicar_a_matriz( vgrad.err_pesos[i], 1/tamaño(conjunto_entrenamiento) )
   Devolver vgrad
```

Algoritmo

```
def learnWithADAM( red, vadam, b1, b2, epsilon, tapren ):
   Para índice l desde 0 hasta red.numero_capas-1: (incluídos):
      Para índice f desde 0 hasta numero_filas( red.pesos[l] ): (incluídos):
         Para índice c desde 0 hasta numero_columnas( red.pesos[l] ): (incluídos):
            momentum = vadam.momento_pesos[l][f][c] / (1 - b1)
            velocity = vadam.velocidad_pesos[l][f][c] / (1 - b2)
            red.pesos[l][f][c] -= (tapren * momentum) / (epsilon + raizcuadrada( velocity ) )
      Para índice i desde 0 hasta tamaño( red.bias[l] ): (incluídos):
         momentum = vadam.momento_bias[l][e] / (1 - b1)
         velocity = vadam.velocidad_bias[l][e] / (1 - b2)
         red.bias[l][e] -= (tapren * momentum) / (epsilon + raizcuadrada( velocity ) )
```

Ejemplo de ejecución:

```
VAdam = Constructor_vadam( Red )
Para época en épocas:
   VGrad = propagación_hacia_detrás( conjunto_entrenamiento, Red, sigmoid, derivative_sigmoid )
   updateAdamVectorValues( Red, VAdam, VGrad, 0.9, 0.99 )
   learnWithADAM( Red, VAdam, 0.9, 0.99, 10^-8, 0.01 )
```

Prueba el código!

Puedes encontrar en este enlace 3 carpetas, cada una con los 3 archivos python de los mismos ejemplos programados en anteriores presentaciones, pero ahora utilizando como algoritmo de optimización ADAM en vez de el descenso de gradiente: https://github.com/Alvaroprueba/1_artificial-neural-networks/tree/main/1-4_adaptative-learning-rate-adam

Descárgalos y ejecuta main.py dentro de cada una de las carpetas.

Cada carpeta tiene cada problema que se intenta resolver en anteriores, pero como se indica, con el optimizador ADAM