



UPC
Universidad Peruana
de Ciencias Aplicadas

REDES NEURONALES:

PERCEPTRÓN MULTICAPA CON BACKPROPAGATION

TA2

Alumna

Ruiz Cerna, Jimena Alexandra

Curso

Programación Concurrida y Distribuida

Sección

CC65

Profesor

Canaval Sánchez, Luis Martín

Ciclo

2020-II

CONTENIDO

INICIO	3
Enlace al Código (Google Colab)	3
Algoritmo seleccionado	3
Descripción	3
IMPLEMENTACIÓN	4
Implementación del algoritmo desde cero usando GO	4
Implementación el algoritmo de manera paralela	5
Uso correcto los mecanismos de sincronización	5
PRUEBAS	6
Selección del dataset y descripción breve.....	6
Carga del dataset en un programa de prueba.....	6
Ejecución del algoritmo implementado con el dataset.....	7
RECURSOS	9

INICIO

Enlace al Código (Google Colab)

<https://bit.ly/3kNoNWp>

Algoritmo seleccionado

Perceptrón multicapa con Backpropagation.

Descripción

Antes de entender el concepto de perceptrón, es bueno comprender la definición de red neuronal y, antes de eso, el de neurona artificial, siendo esta un elemento basado en las neuronas biológicas, donde se tiene elementos de entrada y salida que se procesan en la unidad central. El conjunto de estas neuronas, ordenadas en capas, forma una red neuronal, representación artificial de un cerebro humano, capaz de realizar actividades como la clasificación.

El perceptrón multicapa es un tipo de red neuronal (aprendizaje supervisado) aplicable a problemas donde no se puede realizar una separación lineal de los elementos (como con el perceptrón simple). Se usa varias capas de neuronas en lugar de una sola, pero, para que se note la diferencia, se debe realizar un cambio en la función de activación hacia una no diferenciable, como la función sigmoide. Su aprendizaje se realiza con Backpropagation (Propagación hacia Atrás), de forma que la actualización de pesos se haga con respecto al margen de error entre la respuesta obtenida y la respuesta deseada.

IMPLEMENTACIÓN

Implementación del algoritmo desde cero usando GO

- Bibliotecas utilizadas: fmt, math, sync, time.
- Función de Activación: sigmoide.
- Variables constantes: neuronas, entradas e ítems de entrenamiento.
- Funciones importantes: Para hallar $h[i]$, $dh[i]$ y actualización de pesos.
- Aprendizaje: 0.25
- Épocas: 100

Partes del Algoritmo

1. Asignación: multiplicación de pesos por entradas.

```
h[i] = h[i] + entrenam[l].x[j]*pesos[i][j]
h[neuronas] = h[neuronas] + f[j]*pesos[neuronas][j]
```

2. Cálculo de la salida $f(h[i])$ y $f'(f(h[i]))$ para su posterior uso.

```
f[i] = sigmoid(h[i])
_f[i] = _sigmoid(f[i])
```

3. Cálculo del error y las derivadas.

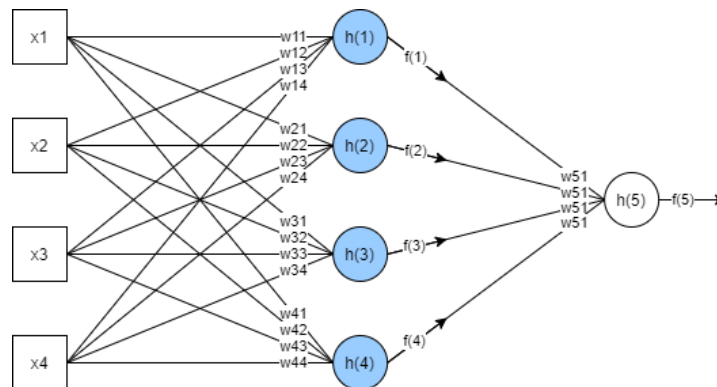
```
err = entrenam[l].z - f[neuronas]
dh[neuronas] = _f[neuronas] * err
dh[i] = _f[i] * pesos[neuronas][i] * dh[neuronas]
```

4. Ajuste de pesos.

```
pesos[neuronas][j] = pesos[neuronas][j] + a*dh[neuronas]*f[j]
pesos[i][j] = pesos[i][j] + a*dh[i]*entrenam[l].x[i]
```

Implementación el algoritmo de manera paralela

El algoritmo del perceptrón multicapa corresponde a la siguiente red neuronal.



La paralelización se realizará en el procesamiento por neurona, a excepción de la neurona de salida, cuyos parámetros dependen de los anteriores (y viceversa en el caso de $d(i)$). El paralelismo se utilizará 3 veces en todo el algoritmo: la primera para la actualización de los parámetros $h[i]$, $f[i]$ y $_f[i]$, la segunda para hallar las derivadas y la tercera para la actualización de pesos. Al tratarse de una red neuronal, es más sencillo tratar a cada neurona como un proceso.

Uso correcto los mecanismos de sincronización

Para la sincronización, se utilizará `WaitGroups`, con sus métodos `Add`, `Wait` y `Done`, donde la variable de sincronización se deberá pasar por referencia a los métodos.

En el proceso de asignación de h , f y $_f$, se necesita que todos los procesos de las primeras 4 neuronas estén terminados, pues la quinta neurona necesita de esa data para su procesamiento individual, es por tal motivo por lo que se coloca el `wg.Wait()` para la comprobación. Lo mismo sucede con la actualización de pesos. El único proceso inverso es el de las derivadas, donde primero se procesa la neurona de salida y luego las de la capa intermedia. El `wg.Wait()` se colocará al final de ese ciclo para asegurar que todos los resultados estén listos para el siguiente proceso. Con respecto a cómo se comprueba que un proceso inició y acabó, antes de usar el `go + función`, se agrega un `wg.Add(1)` y, al terminar los métodos de la función, se envía un `defer wg.Done()`.

```
wg.Add(1)
go hallarD(i, &wg)
```

```
func hallarD(i int, wg *sync.WaitGroup) {
    dh[i] = _f[i] * pesos[neuronas][i] * dh[neuronas]
    defer wg.Done()
}
```

PRUEBAS

Selección del dataset y descripción breve

Se eligió The Iris Dataset, originalmente llamado UCI Machine Learning Repository: Iris Data Set, por sus variables de entrada cualitativas (largo y ancho del pétalo y sépalo, contando 4 entradas) y sus tres modos de clasificación con respecto a la salida (setosa, versicolor y virginica). Su función es la de otorgar información con respecto a la clasificación por especies de la flor iris.

	sepal_length	sepal_width	petal_length	petal_width	species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa

Carga del dataset en un programa de prueba

El dataset se descargó en formato csv, contando con 150 registros de 50 cada especie, desde la siguiente url pública

<https://gist.github.com/curran/a08a1080b88344b0c8a7>

Se implementó un programa para su lectura (prueba y validación) llamado *rcsv.go*.

- Bibliotecas utilizadas: encoding/csv, io, log, os, strconv.
- Funciones importantes:
 - Lectura de cada registro como entrada por separado.
 - Parse de los strings a float64.
 - Conversión de la respuesta de especies en números.
 - Se invoca desde pmv2.go.

Se implementó un programa para su prueba llamado *prueba.go*.

- Bibliotecas utilizadas: fmt, math.
- Funciones importantes:
 - Lectura de los pesos luego del entrenamiento (ejecución del algoritmo).
 - Halla los resultados con la función de activación.
 - Compara los resultados con puntos de corte establecidos luego de la observación y análisis de pesos y resultados probables.
 - Detalla el informe con la clasificación y el resultado

- Datos de prueba:

```
// setosa
hallaF(5.2, 3.6, 1.5, 0.3)
hallaF(5.1, 3.2, 1.6, 0.3)
hallaF(4.8, 3.3, 1.4, 0.3)
hallaF(4.7, 3.2, 1.6, 0.3)

// versicolor
hallaF(7.1, 3.3, 4.8, 1.5)
hallaF(6.5, 3.3, 4.6, 1.6)
hallaF(7.0, 3.2, 5.0, 1.6)
hallaF(5.6, 2.4, 4.1, 1.4)

// virginica
hallaF(6.4, 3.4, 6.1, 2.6)
hallaF(5.9, 2.8, 5.2, 2.0)
hallaF(7.2, 3.1, 6.0, 2.2)
hallaF(6.4, 3.0, 5.7, 1.9)
```


Ejecución del algoritmo implementado con el dataset

La clasificación (prueba) con el algoritmo y dataset funciona bajo los siguientes pasos (debido al parámetro época, pesos iniciales y procesador)

1. Ejecutar la sobre-escritura del archivo pesos.txt (para que empiece con los pesos establecidos).


Pesos

 MOSTRAR CÓDIGO

 Overwriting pesos.txt

2. Ejecutar el algoritmo solo una vez (go run pmv2.go rw.go rcsv.go)

Ejecutando el algoritmo

 !go run pmv2.go rw.go rcsv.go

Pesos Finales:

Neurona # 1 :	Neurona # 2 :	Neurona # 3 :	Neurona # 4 :	Neurona # 5 :
w 1 1 : -0.698	w 2 1 : 0.961	w 3 1 : 1.002	w 4 1 : -0.32	w 5 1 : 6.044
w 1 2 : -1.498	w 2 2 : 1.161	w 3 2 : 0.602	w 4 2 : 0.68	w 5 2 : -6.754
w 1 3 : -0.298	w 2 3 : 0.461	w 3 3 : 1.102	w 4 3 : 0.88	w 5 3 : -5.85
w 1 4 : -0.498	w 2 4 : 0.511	w 3 4 : -0.198	w 4 4 : 0.28	w 5 4 : 14.667

Duración del programa: 81.669478ms

3. Ejecutar el test (go run test.go rw.go)

```
!go run test.go rw.go

Respuesta: setosa 0.20996514384779794
Respuesta: setosa 0.2527314710248583
Respuesta: setosa 0.19767015778306982
Respuesta: setosa 0.2675500414901454
Respuesta: versicolor 0.9277506004150696
Respuesta: versicolor 0.9250316477893795
Respuesta: versicolor 0.9331332411340003
Respuesta: versicolor 0.9107548212948537
Respuesta: virginica 0.9486496712215214
Respuesta: virginica 0.9406418595629267
Respuesta: virginica 0.9471017586694086
Respuesta: virginica 0.9444987594969353

Exactitud con datos de prueba: 100 %
```

Con respecto a los ejemplos puestos en el programa de prueba, se tiene un 100% de exactitud de clasificación para cada tipo con respecto al perceptrón multicapa entrenado (con un programa de menos de un segundo de duración) con el dataset iris de 150 registros.

RECURSOS

- Mejía, J. (2004). *Sistema de detección de intrusos en redes de comunicaciones utilizando redes neuronales* [Tesis]. Capítulo 3: *Perceptrón Multicapa*. Universidad de Las Américas Puebla. Recuperado de: <https://bit.ly/34JcPav> [Consulta: 31 de octubre del 2020]
- Bodnar, J. (2020). *Go read file*. ZetCode. Recuperado de: <https://bit.ly/35T4RuX> [Consulta: 31 de octubre del 2020]
- Raina, A. (2018). *Reading a simple CSV in Go*. Recuperado de: <https://bit.ly/2TJ771W> [Consulta: 31 de octubre del 2020]
- McGranaghan, M. (s.f.) *Go by Example: WaitGroups*. Recuperado de: <https://bit.ly/381fnCX> [Consulta: 31 de octubre del 2020]
- Pérez, A. (2019). *Concurrencia en Golang: WaitGroups*. Friends of Go. Recuperado de: <https://bit.ly/3oK6lQF> [Consulta: 31 de octubre del 2020]