UPC
Universidad Peruana
de Ciencias Aplicadas

API DISTRIBUIDO PARA PERCEPTRÓN MULTICAPA

TRABAJO FINAL

Integrantes
Bravo Berrio, Anthony Joaquín
Ruiz Cerna, Jimena Alexandra

Curso
Programación Concurrida y Distribuida

Sección
CC65

Profesor
Canaval Sánchez, Luis Martín

Ciclo
2020-II

INTRODUCCIÓN, DESCRIPCIÓN DEL PROBLEMA Y MOTIVACIÓN

La clasificación de especies ha sido un problema recurrente, sobre todo el campo de la biología y los procesos industriales. En el primero, se busca tener llevar un registro de todas las variaciones de flora mientras que, en el segundo, se busca que ciertos productos de origen vegetal cumplan con los estándares adecuados para pasar a la etapa de producción. Los dos tienen en común la necesidad de procesas ciertas entradas cualitativas o cuantitativas para tomar decisiones sobre clasificación relacionado a las plantas o alimentos que provienen de estas.

Se realizó una validación con ayuda de una estudiante de Ingeniería Industrial, donde se aprecia que incluso este concepto puede ser llevado a la medición de lípidos en algas para comprobar si son del tipo correcto para el consumo humano.

Enlace a la entrevista: https://youtu.be/nHtEzhHJV-U

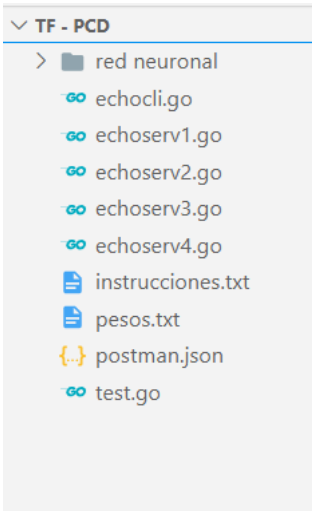
Tras la implementación previa de un perceptrón multicapa capaz de clasificar al a una flor iris en "setosa", "virginica" o "variocolor" según el ancho y largo de su sépalo y pétalo, se desea compartir ahora esa red capaz de realizar tal acción mediante un API. La red mencionada, de 4 nodos en su capa intermedia, poseía un programa de prueba que se mostró en el TA2 (donde también se utilizaba concurrencia para el procesamiento de cada neurona), pero su acceso mediante comando era complicado para un usuario que solo busca clasificar plantas, por lo que se efectuó el reemplazo mencionado.

La motivación de estructurar este API de manera distribuida es el balanceo de la carga, dividido ahora en 4 servidores que responden a otro servidor que actúa de cliente que, a su vez, estructura la respuesta adecuada al usuario y maneja los endpoints. De esta forma, la aplicación soportaría múltiples pedidos.

OBJETIVOS

- El objetivo principal del proyecto es la creación de un Api capaz de otorgarle al usuario la clasificación de la flor iris de acuerdo con los parámetros que este le envía. Este resultado es obtenido mediante una red neuronal previamente entrenada con el conjunto de datos iris de Fisher.
- Además, la solución incluye contingencia ante gran demanda de usuarios, utilizándose cuatro nodos servidores gestionados por un nodo cliente que se encarga de balancear la carga entre estos.
- El API incluye respuesta antes pedidos tipo get por parámetro y post por json.
- El API responde con json indicando en qué nodo ha sido procesada la información.

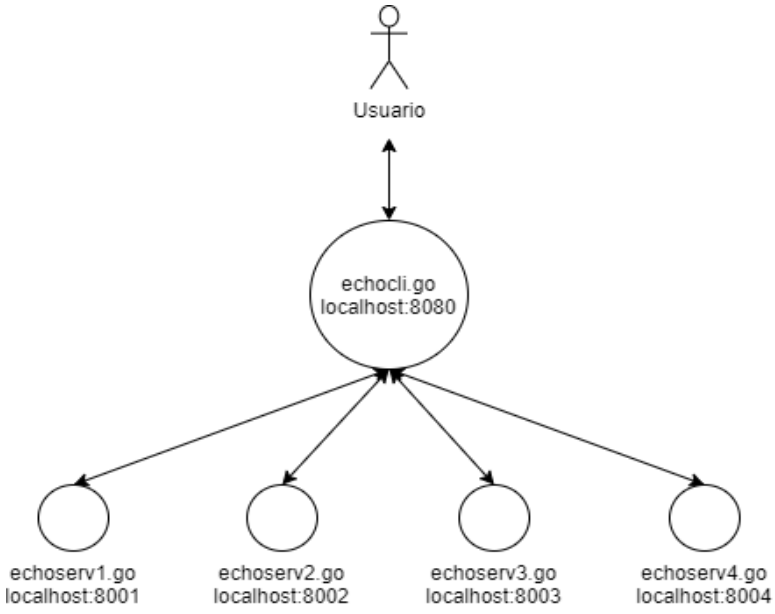
DISEÑO



Los componentes de la solución se componen de varios archivos de lectura, programas y la red neuronal. Como se explicó anteriormente, se trabajará con la red ya entrenada, por lo que no será necesario volver a ejecutar los programas propios del perceptrón, pero sí será necesario el archivo de pesos obtenidos. También se cuenta con un archivo json correspondiente a la colección creada en PostMan y un archivo de texto con las instrucciones, incluida la instalación de una biblioteca externa.

Centrándose en el API, se tiene a los cinco nodos distribuidos en los programas "echocli.go", "echoserv1.go", "echoserv2.go", "echoserv3.go"y "echoserv4.go", que se deberán ejecutar en simultáneo para el correcto funcionamiento.

Con respecto a la arquitectura, se tiene a "echocli.go" como nodo cliente se interacción directa con el usuario. Los programas "echoserv" son los servidores que actuarán según "echocli" se los pida (ya que, además de funcionar como controller, hace de balanceador de carga).



DESARROLLO

En primer lugar, se crearon los nodos servidores, capaces de recibir archivos tipo csv y procesarlos (clasificarlos) mediante la función que, anteriormente, estaba alojada en el programa "test.go".

```
func hallaF(x1, x2, x3, x4 float64) string {
    f1 := sigmoid(x1*pesos[1][1] + x2*pesos[1][2] + x3*pesos[1][3] + x4*pesos[1][4])
    f2 := sigmoid(x1*pesos[2][1] + x2*pesos[2][2] + x3*pesos[2][3] + x4*pesos[2][4])
    f3 := sigmoid(x1*pesos[3][1] + x2*pesos[3][2] + x3*pesos[3][3] + x4*pesos[3][4])
    f4 := sigmoid(x1*pesos[4][1] + x2*pesos[4][2] + x3*pesos[4][3] + x4*pesos[4][4])
    fy := sigmoid(f1*pesos[5][1] + f2*pesos[5][2] + f3*pesos[5][3] + f4*pesos[5][4])

    if fy < 0.45 {
        return "setosa"
    }
    if fy < 0.575 && fy >= 0.45 {
        return "versicolor"
    }
    if fy >= 0.575 {
        return "virginica"
    }
    return ""
```

Luego, se codificó a la función main para que escuchara la llamada del servidor principal. Al enviarse un mensaje desde el nodo cliente, se activaría la función que decodifica el mensaje, procesa las entradas y devuelve un json. Este proceso se repite para los 4 nodos servidores, solo que con variación en el puerto del localhost.

```
func main() {
    ln, _ := net.Listen("tcp", "localhost:8001")
    defer ln.Close()
```

```
    y := hallaF(aux[1], aux[2], aux[3], aux[4])

    data := &Response{}
    data.LargoSepalo = aux[1]
    data.AnchoSepalo = aux[2]
    data.LargoPetalo = aux[3]
    data.AnchoPetalo = aux[4]
    data.Localhost = "8001"
    data.Clasificacion = y

    responseJSON, _ := json.Marshal(data)
    fmt.Fprint(con, string(responseJSON)+"\n")
```

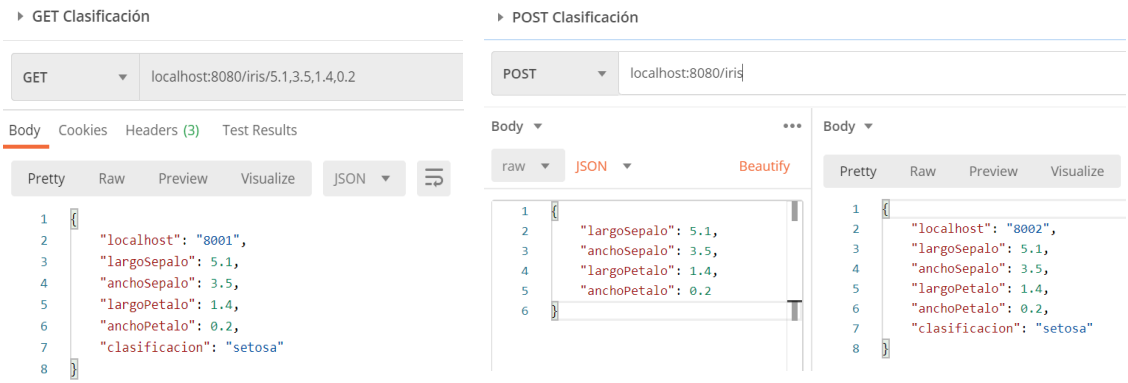
En el nodo principal, primero se crearon las rutas para los endpoints, además de las funciones respectivas a las que invocan.

```
func main() {
    router := mux.NewRouter().StrictSlash(true)
    router.HandleFunc("/", homeLink)
    router.HandleFunc("/iris", postClasification).Methods("POST")
    router.HandleFunc("/iris/{parameters}", getClasification).Methods("GET")
    log.Fatal(http.ListenAndServe(":8080", router))
```

La lectura es diferente para ambos endpoints, pero sí tienen en común el sistema de distribución de carga, que es un contador global que va dando turnos a los servidores.

```
if cont == 1 {
    fmt.Fprint(con1, msg)
    cont++
    msg, _ = r1.ReadString('\n')
} else if cont == 2 {
    fmt.Fprint(con2, msg)
    cont++
    msg, _ = r2.ReadString('\n')
} else if cont == 3 {
    fmt.Fprint(con3, msg)
    cont++
    msg, _ = r3.ReadString('\n')
} else if cont == 4 {
    fmt.Fprint(con4, msg)
    cont = 1
    msg, _ = r4.ReadString('\n')
}
```

Las pruebas desde Postman muestran el mismo resultado, ya sea pasando las entradas por parámetro separadas por comas o mediante un json.



Enlace al vídeo de prueba: https://youtu.be/bCCaHf0-EUE

CONCLUSIONES

- El usuario pude obtener la clasificación de la flor iris con solo 4 parámetros: largo y ancho del sépalo y largo y ancho del pétalo.
- Se usa programación distribuida con balanceo de carga, conexión entre servidores y lectura de request tipo json y parámetros.
- Se obtiene una respuesta en formato json con la información ingresada, el nodo donde se procesó la información y la clasificación.
- Se propone una posterior implementación para otro tipo de plantas o productos de origen vegetal.

REFERENCIAS

Algorithms to Go. (s.f.). Convert between float and string. Recuperado de: https://yourbasic.org/golang/convert-string-to-float/ [Consulta: 25 de noviembre del 2020]

Friends of Go. (2019). *¿Cómo crear una API Rest en Golang?* Recuperado de: https://blog.friendsofgo.tech/posts/como_crear_una_api_rest_en_golang/ [Consulta: 25 de noviembre del 2020]

Golang.org. (s.f.). *Package JSON*. Recuperado de: https://golang.org/pkg/encoding/json/ [Consulta: 25 de noviembre del 2020]

Gorilla Web Toolkit. (2020). *gorilla/mux*. [Repositorio de código]. Recuperado de: https://github.com/gorilla/mux [Consulta: 25 de noviembre del 2020]

TutorialEdge.net. (2017). *Creating a RESTful API With Golang*. Recuperado de: https://tutorialedge.net/golang/creating-restful-api-with-golang/ [Consulta: 25 de noviembre del 2020]