Sistemas de Inteligencia Artificial

Trabajo Practico de Redes Neuronales

Grupo 12

Alan Arturo Hernández Gutiérrez	alhernandez@itba.edu.ar	60438
Kevin Chidiac	kchidiac@itba.edu.ar	60431
Romain Thibaut Marie Latron	rlatron@itba.edu.ar	60440
Ramiro Hernán Olivera Fedi	rolivera@itba.edu.ar	56498

Índice

Introducción

En el desarrollo del siguiente trabajo práctico se implementa un algoritmo que permita construir perceptrones multicapa a partir de parámetros obtenidos desde un archivo de configuración. El algoritmo, escrito en Octave, debe ser capaz de generar una arquitectura determinada, donde pueden ser parametrizadas la cantidad de capas de la red, sus neuronas, las funciones de activación, los bias individuales, optimizaciones al algoritmo de backpropagation, épocas a ejecutar, valores iniciales de etas, etc…

Se busca además, probar distintas arquitecturas y parámetros para definir la mejor arquitectura para *aprender* el dataset *terrain12.*

Desarrollo

El trabajo puede dividirse a grandes rasgos en 3 etapas de desarrollo:

Parseado y lectura de configuración

Se decidió utilizar un archivo JSON como archivo de configuración del algoritmo, debido a que se trata de un formato altamente portable, lo cual lo hace ideal para clarificar la configuración. El formato del archivo *config.json* es el siguiente:

`layers`	Lista de capas donde cada capa tiene una función de activación `activation`, un `bias` y una cantidad de neuronas `neurons`
`optimization`	Lista de algoritmos de optimización de backpropagation. Cada algoritmo es de un tipo `type` y tiene un objeto de parámetros `params`
`training`	Path del training set
`test`	Path del test set
`error`	Función de costo
`epochs`	Número de épocas a correr el entrenamiento
`eta`	Valo de eta inicial

El archivo de configuración es posteriormente parseado en Octave utilizando la librería Open Source `jsonlab`[1], y cargado en una estructura para ser accedida a lo largo de la ejecución del programa.

Construcción del algoritmo genérico

Se construyó el algoritmo genérico como función de los parámetros del archivo de configuración, haciendo foco en la separación y reutilización de código, a través de un diseño modular. Así por ejemplo, el módulo `activation` mapea los nombres de las funciones de transferencia a las funciones reales.

El archivo `network.m` contiene la implementación del algoritmo. Sus métodos principales son:

`build`	Setea los pesos de la red con valores aleatorios
`train`	Entrena a la red de manera estocástica iterando por las épocas, los patrones, y aplicando a cada uno `feed-forward` y `backpropagation`. A su vez, analiza si la configuración contiene optimizaciones, itera sobre ellas y las ejecuta.
`test`	Corre un set de patrones sobre la red, sin alterar los pesos, y devuelve el error.

Lectura y normalizado de datos

Para la lectura de datos, se decidió también tomar como input un archivo JSON. Es por esta razón que se decidió crear un script de `javascript` que transforma el input del terreno en un archivo con el siguiente formato:

`data`	Lista de elementos del tipo `patrón`
`patrón`	Elemento con un `output` y un `input`
`output`	Lista de valores
`input`	Lista de valores

En todo momento se asume que el formato del archivo de entrada es válido. Una vez más, se parsea el archivo utilizando la librería `jsonlab`. Dado que el dominio de los datos puede estar disperso, se normalizaron los valores tanto de entrada y de salida al intervalo [-1; 1] manteniendo la distribución de los mismos.

Análisis de arquitecturas

Se presenta a continuación el detalle de las distintas arquitecturas probadas con su correspondiente configuración. Nótese que solo se muestran los campos importantes, pero que el archivo de configuración aún debe tener los otros. Si bien hay infinitas arquitecturas

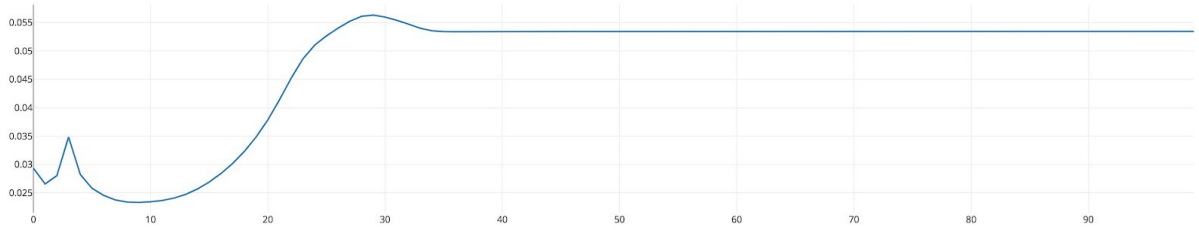
posibles, se muestran aquellas cuyos parámetros se modificaron en las variables más significativas. Así pues se muestran las distintas variaciones en la cantidad de neuronas en la capa oculta, las funciones de activación, el número de épocas, y las distintas mejoras al algoritmo de `backpropagation`.

Todas las pruebas se hicieron con la misma inicialización de pesos aleatorios, utilizando como semilla el valor constante `42`. De esta forma, se puede apreciar mejor los cambios en la eficiencia y efectividad de las distintas arquitecturas.

El primer caso a analizar consta de una sola capa oculta de 5 neuronas, más el `bias`. Podemos ver con facilidad que esta arquitectura no permite la correcta representación de los datos de manera interna, provocando que se pierda información.

```
{
  "layers": [{ "activation": "ID", "bias": 1, "neurons": 2 }, {
"activation": "TANH", "bias": 1, "neurons": 5 }, { "activation":
"TANH", "bias": 1, "neurons": 1 }],
  "epochs": 100,
  "eta": 0.1,
  ...
}
```

Evolución del error cuadrático medio



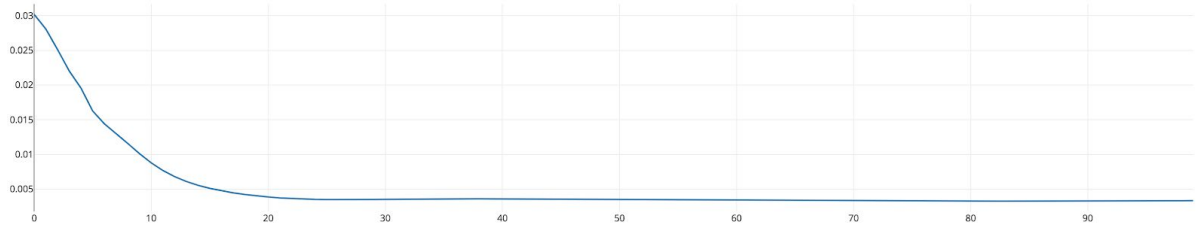
Error cuadrático medio del test set	`0.061336`
Tiempo de ejecución	`30.510 segundos`

Para el segundo caso se decidió agregar 5 neuronas más a la capa oculta. En este ejemplo, podemos ver que 10 neuronas en una sola capa oculta son suficientes para representar de manera bastante certera el dataset del terreno.

```
{
```

```
  "layers": [{ "activation": "ID", "bias": 1, "neurons": 2 }, {
"activation": "TANH", "bias": 1, "neurons": 10 }, { "activation":
"TANH", "bias": 1, "neurons": 1 }],
  "epochs": 100,
  "eta": 0.1,
  ...
}
```

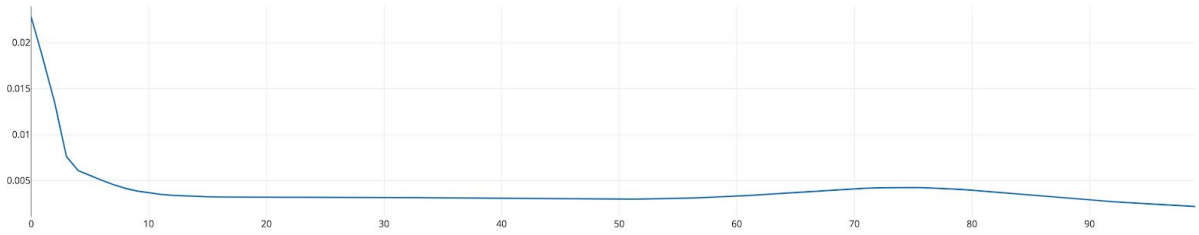
Evolución del error cuadrático medio



Error cuadrático medio del test set	`0.0039753`
Tiempo de ejecución	`31.984 segundos`

Continuando con la misma estrategia duplicamos la cantidad de neuronas de la capa oculta hasta alcanzar las 20 neuronas más el `bias`. Bajo esta arquitectura puede apreciarse cómo a mayor cantidad de neuronas, menor cantidad de épocas son necesarias para reducir el error.

```
{
  "layers": [{ "activation": "ID", "bias": 1, "neurons": 2 }, {
"activation": "TANH", "bias": 1, "neurons": 20 }, { "activation":
"TANH", "bias": 1, "neurons": 1 }],
  "epochs": 100,
  "eta": 0.1,
  ...
}
```

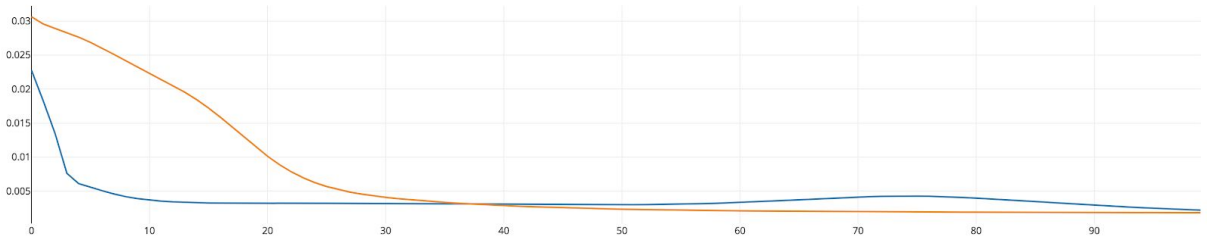


Error cuadrático medio del test set	`0.0022344`
Tiempo de ejecución	`29.579 segundos`

Continuemos entonces utilizando una capa oculta de 20 neuronas más el `bias` y comparemos esa arquitectura con una similar, pero cuya función de activación en la capa oculta sea la exponencial. Se graficó en naranja la evolución del error en la nueva arquitectura, y en azul la evolución con la arquitectura anterior.

```
{
  "layers": [{ "activation": "ID", "bias": 1, "neurons": 2 }, {
"activation": "EXP", "bias": 1, "neurons": 20 }, { "activation":
"TANH", "bias": 1, "neurons": 1 }],
  "epochs": 100,
  "eta": 0.1,
  ...
}
```

Como puede apreciarse la función `tanh` parece converger mucho más rápido para un corto número de iteraciones. No obstante, notemos que en un número intermedio de épocas, es la función `exp` la que reduce en mayor cantidad el error.

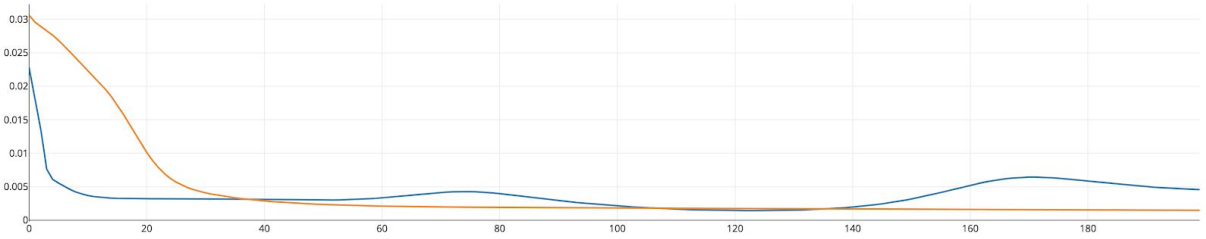


Error cuadrático medio del test set	`0.0024315`
Tiempo de ejecución	`31.530 segundos`

Probemos ahora entonces duplicar el número de épocas y grafiquemos nuevamente las dos arquitecturas: En naranja la que tiene en su capa oculta como función de transferencia a la exponencial, y en azul la que tiene tangente hiperbólica.

```
{
```

```
  "layers": [{ "activation": "ID", "bias": 1, "neurons": 2 }, {
"activation": "EXP", "bias": 1, "neurons": 20 }, { "activation":
"TANH", "bias": 1, "neurons": 1 }],
  "epochs": 200,
  "eta": 0.1,
  ...
}
```

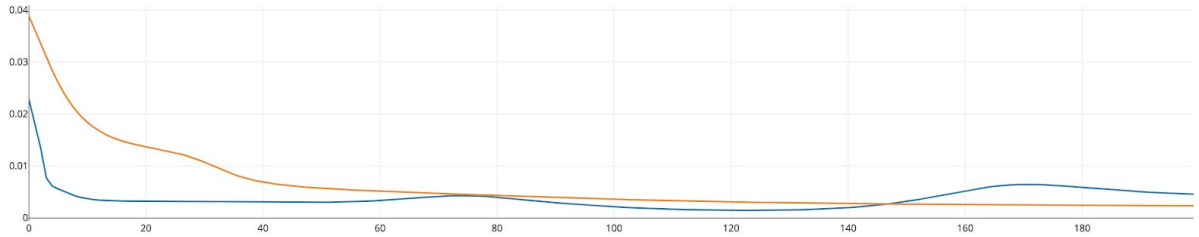


Notemos como la tangente hiperbólica comienza a aumentar, presumiblemente debido al gran tamaño del `eta` inicial, mientras que la función exponencial, parece converger hacia el error nulo.

	`exp`	`tanh`
Error cuadrático medio del test set	`0.0018896`	`0.0018896`
Tiempo de ejecución	`57.517 segundos`	`57.791 segundos`

Para verificar nuestra hipótesis, utilicemos la función `tanh` y reduzcamos el `eta` inicial a `0.01`. Comparemos esta nueva parametrización con otra con la misma función de transferencia y número de épocas, pero un `eta` de `0.1`.

```
{
  "layers": [{ "activation": "ID", "bias": 1, "neurons": 2 }, {
"activation": "TANH", "bias": 1, "neurons": 20 }, { "activation":
"TANH", "bias": 1, "neurons": 1 }],
  "epochs": 200,
  "eta": 0.01,
  ...
}
```



En naranja se muestra la arquitectura con `eta 0.01` y en azul aquella con `0.1`. Al parecer nuestra hipótesis era correcta. Notemos como la función naranja se acerca más lentamente al 0 pero sin tantas fluctuaciones.

	`eta = 0.01`	`eta = 0.1`
Error cuadrático medio del test set	`0.0012648`	`0.0018896`
Tiempo de ejecución	`1:25.25 minutos`	`1:23.03 minutos`

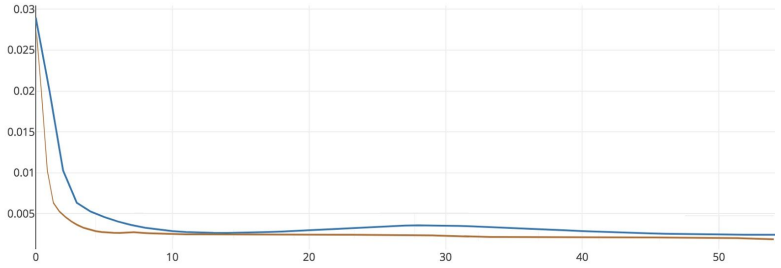
Probemos ahora `eta mejorado`, una de las mejoras al algoritmo de `backpropagation`. Este algoritmo consiste básicamente de modificar el valor del `eta` de acuerdo a qué tan bien se está desempeñando la red neuronal (medida por la función de costo). La parametrización de este algoritmo consiste en:

`k`	Cantidad de pasos en los que se reduce el error para sumar `alpha` al `eta`
`alpha`	Valor a sumarle al `eta` luego de que la red haya mejorado sus resultados consistentemente (medidos por `k`).
`beta`	Tasa de reducción del `eta` cuando el error aumenta en un paso.

Los resultados son poco favorecedores. Si se logró conseguir los valores que ofrecieron los mejores resultados, no obstante, el tiempo aumentó significativamente y la optimización no se vió reflejada en la reducción del error.

`eta inicial`	`k`	`alpha`	`beta`
`0.8`	`2`	`0.05`	`0.8`

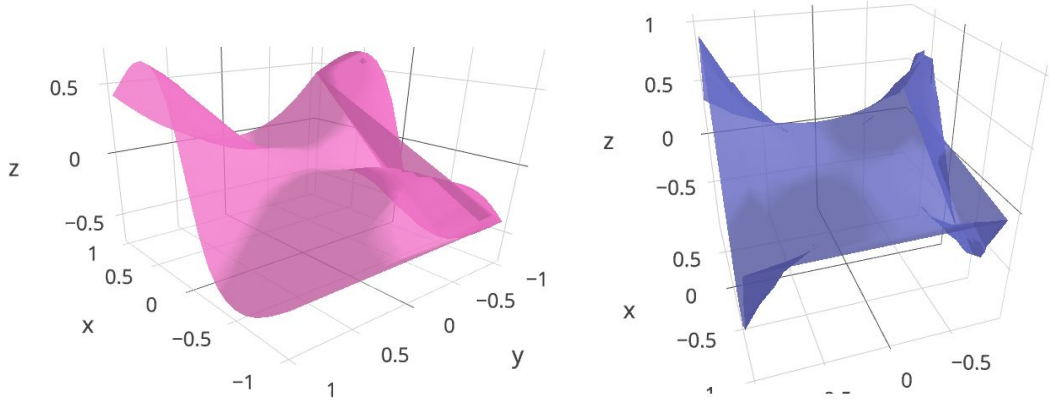
Probemos ahora `momentum`, otra de las mejoras al algoritmo de `backpropagation`. La idea del algoritmo es desestimar las variaciones en el error, y en su lugar dejarse guiar por la pendiente de la recta. Se utilizó como parámetro `alpha` el valor recomendado `0.9`.



Los resultados son favorecedores contra la falta de optimizaciones.

	`con momentum`	`sin momentum`
Error cuadrático medio del test set	`0.0011449`	`0.0012648`
Tiempo de ejecución	`1:22.35 minutos`	`1:25.25 minutos`

Hasta el momento la arquitectura que mejor se desempeñó fue TAL. Generamos un conjunto de puntos equidistantes en el plano `[-1, 1] x [-1, 1]` espaciado por pasos de `0.1` y lo alimentamos a la red entrenada con la arquitectura mencionada para obtener una aproximación al terreno. A continuación se muestra graficada dicha aproximación en color rosa. Junto a este gráfico, se encuentra la representación gráfica de los valores que componen el archivo *terrain12.*



Notemos la similitud de las curvas, brindando una visualización intuitiva de la efectividad de la arquitectura seleccionada. Desestimamos la diferencia entre la *suavidad* de las curvas debido a la diferencia entre los valores de entrada con los que alimentamos la red, ya que en el caso de la curva rosa, ésta fue alimentada con valores equidistantes del plano.

Mejoras futuras

Cómo futuras mejoras al trabajo consideramos:

- Inicialización de pesos

  Agregar al archivo de configuración un parámetro que permita inicializar la red con valores predeterminados.

- Modos de entrenamiento

  Parametrizar el modo de entrenamiento, permitiendo no solo entrenamientos incrementales, sino también batch y mini-batch.

Conclusion

El trabajo permitió al equipo desarrollar su entendimiento de redes neuronales mediante el análisis de un caso práctico.

Por lo expuesto y analizado en la sección correspondiente, podemos notar que se logró construir una arquitectura que permite representar los datos con la que se la entrenó, y generaliza particularmente bien, obteniendo errores $MSE < 1.1 * 10^{-3}$.

Dicha arquitectura utiliza los siguientes parámetros de configuración:

```
{
  "layers": [{ "activation": "ID", "bias": 1, "neurons": 2 }, {
"activation": "TANH", "bias": 1, "neurons": 20 }, { "activation":
"TANH", "bias": 1, "neurons": 1 }],
  "epochs": 200,
  "eta": 0.01,
  "optimization": [{"type": "MOMENTUM", "params": { "alpha": 0.9
}}],
  ...
}
```