Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-5438 Inteligencia Artificial II
Trimestre Septiembre - Diciembre 2023

Proyecto 2

Estudiantes:

Figueira, Adelina

Bandez, Jesús

Carnets:

15-10484

17-10046

Profesor

Carlos Infante

24 de noviembre de 2023

Detalles de la implementación

Se implementó el algoritmo de propagación hacia atrás con ayuda de matrices para facilitar los cálculos. Se crearon clases para las capas, capa de salida, un objeto que se encarga de manejar las operaciones sobre las capas ocultas y un objeto que representa a toda la red haciendo uso de las clases anteriores.

1. Layer: Es una clase que representa a las capas de la red, en cada iteración una capa tiene una matriz de pesos W, un vector de valores de activación A, un vector de bias B y dos valores bias_gradient y weight_gradient que representan los gradientes locales en la capa.

2. Layer_Output: Subclase de Layer que maneja las operaciones que se realizan sobre la capa, como la propagación del input, gradiente y la actualización de los pesos.

3. Hidden_Layers: Es una clase que se encarga de guardar todas las capas ocultas que existen en la red, contiene una lista de layers en donde se alojan objetos de tipo Layer y una lista ini en donde se alojan los valores de activación anteriores. Esta clase maneja las operaciones de propagación de input y gradientes sobre las capas ocultas y actualiza los pesos de dichas capas.

4. Network: Es una clase que representa a la red neuronal. Contiene todos los métodos necesarios para hacer el entrenamiento de la red, un método predict para predecir valores y se inicializa con los siguientes valores:

    - neurons_per_layer : Es una lista de tuplas que representa los tamaños de las matrices en cada una de las capas. Al ingresar los valores la primera tupla debe tener como primer valor la cantidad de columnas de X, luego el último valor de la tupla debe ser el mismo que el primer valor de la tupla siguiente, por ejemplo: [(2,3),(3,4),(4,1)]. El último valor de la última tupla representa la cantidad de columnas que existen en Y, si es un clasificador binario debe ser 1, si es multiclase para 3 clases debe ser 3.
    - g: Es la función logística.
    - epsilon: Es el valor que controla si el error es lo suficientemente pequeño como para acabar el ciclo.

- alpha: Es el learning rate del modelo.

La función logística utiliza fue la siguiente:

```
def g(x):
    a = 1
    return 1/(1+np.exp(-x))
```

Además se utilizó la derivada de la función logística gp:

```
def gp(x):
    return (x*(1-x))
```

¿Cómo se entrena a la red neuronal?

Primero se inicializa la red neuronal. Se crean las capas ocultas, agregando instancias de la clase Layer en Hidden_Layer basándose en una lista de tuplas que contienen la cantidad de neuronas por cada capa. Luego se crea una instancia de la capa de salida Layer_Output con los valores introducidos por el usuario.

```
for x,y in self.neurons_per_layer[:-1]:
    self.hidden_layers.add_layer(Layer(x,y))
x,y = self.neurons_per_layer[-1]
self.layer_output = Layer_Output(x,y)
```

Luego se itera i veces por los siguientes pasos, hasta alcanzar minimizar el error o hasta que terminen las iteraciones.

1. Forward Propagation (Propagación hacia adelante)
2. Backwards Propagation (Propagación hacia atrás)
3. Actualización de pesos

Forward Propagation

Este paso se encarga de propagar los inputs, primero por las capas ocultas y luego en la capa de salida. Con el forward propagation se toma el valor de salida de la capa anterior y se multiplica por los pesos más el bias de la capa actual, luego a esto se le aplica la función logística para obtener la activación de la capa y finalmente este valor se pasa a la capa siguiente.

```
for layer in self.layers:
    self.ini.append(a_0)
    res = (a_0 @ layer.W) + layer.B
    layer.A = f(res)
    a_0 = layer.A
```

El valor inicial es la matriz X que se desea entrenar. Finalmente, la última capa oculta retorna su valor de activación y esta será la entrada para la propagación de la capa de salida.

```
res = (a0 @ self.W) + self.B
self.A = f(res)
```

Backwards Propagation

Como ya se conocen los valores de activación para cada capa, es momento de propagar los gradientes de una capa sobre las capas anteriores. Como se realiza desde la capa final a la inicial, se comienza por la capa de salida. Se creó un método llamado propagate_deltas_output que recibe como entrada la última activación de las capas ocultas y el valor del vector de salida con los resultados deseados, Y.

```
def propagate_deltas_output(self, previous_activation, Y):
        resta = Y-self.A
        error = (resta.T @ resta)/len(Y)
        error_gradient = (self.A - Y)*2/len(Y)
        layer_gradient = error_gradient @ self.W.T
        self.weight_gradient = previous_activation.T @ error_gradient
        self.bias_gradient = np.sum(error_gradient, axis=0, keepdims=True)
```

```
        self.error = error
        return layer_gradient
```

La propagación de la capa de salida calcula el error cuadrático medio con una multiplicación matricial que se aloja en la variable error. Luego se calcula el gradiente de la capa actual en la variable layer_gradient que es calculado al multiplicar la transpuesta de la matriz de pesos actual por la matriz que se obtiene al restar los valores de activación de la capa de salida con Y, todo esto multiplicado por 2 y dividido por el tamaño de Y. Esto es el equivalente a usar la derivada de la función logística g. Finalmente, se actualiza el gradiente de los pesos y bias y el error de la capa.

La variable layer_gradient de la última capa será la entrada de la función propagate_deltas_on_layers que se encarga de propagar el gradiente sobre todas las capas ocultas y calcula los gradientes locales de cada una de las capas.

```
def propagate_deltas_on_layers(self,upstream_gradient):
        activation_derivative = gp
        for layer,ini in reversed(list(zip(self.layers,self.ini))):
            activation_gradient = activation_derivative(layer.A) *
            ↪  upstream_gradient
            layer_gradient = activation_gradient @ layer.W.T
            weight_gradient = ini.T @ activation_gradient
            bias_gradient = np.sum(activation_gradient, axis=0, keepdims=True)
            layer.weight_gradient = weight_gradient
            layer.bias_gradient = bias_gradient
            upstream_gradient = layer_gradient
```

Para las capas ocultas se hace uso de la función gp, que es la derivada de la función logística g. Primero se calcula la variable activation_gradient que es la función logística evaluada con los valores de activación de la capa actual multiplicada por el gradiente de la capa anterior, luego se obtiene el gradiente de la capa actual, multiplicando este valor por la transpuesta de

los pesos. Finalmente, se actualizan los gradientes de los pesos con ayuda de los valores de las activaciones de cada una de las capas anteriores y el activation_gradient.

Actualización de los pesos

Una vez se hayan propagado los inputs y los gradientes, se puede realizar la actualización de los pesos de cada una de las capas. Esta actualización puede realizarse en cualquier orden, se realizó primero la actualización de la capa de salida y finalmente las capas ocultas.
En ambos casos, la actualización de pesos se realiza de manera sencilla, utilizando el valor alpha o tasa de aprendizaje y el gradiente. También se actualiza el valor del vector de bias que se utiliza para los cálculos durante la propagación de inputs.

```
self.W = self.W - alpha*self.weight_gradient
self.B = self.B - alpha*self.bias_gradient
```

Como se mencionó anteriormente, estos pasos se repiten reiteradamente hasta acabarse las iteraciones o hasta que la norma del vector de error de la capa de salida, sea menor al valor de epsilon que se introduce al inicializar la red neuronal.

Clasificador

Ambos clasificadores toman los valores de alpha, epsilon, las iteraciones y las capas ocultas. Los clasificadores toman una lista de enteros que representa la cantidad de neuronas por cada capa oculta y genera los valores de X e Y basándose en la lectura del archivo CSV con los datos preprocesados.

Clasificador Binario

Los valores de X son las columnas del archivo que representan las características de cada clase de la planta de género Iris y se representa como una matriz. Los valores de Y son la salida de la clase y representan si la planta es o no de dicha clase, por lo que es un vector de 1 y 0s.

Clasificador Multiclase

De manera similar al clasificador multiclase, se toman los valores de X como una matriz que representa las características de cada clase y Y es una matriz que incluye vectores columna de 1 y 0s que representan a cada una de las clases 'Iris-setosa', 'Iris-virginica' e 'Iris-versicolor'.

Preprocesamiento de los datos

Se normalizaron todas las columnas del archivo iris.csv: 'sepal_length', 'sepal_width', 'petal_length', 'petal_width'. Como el archivo cuenta con 150 registros y cada especie tiene 50 filas, se tomaron 10 registros de cada especie para realizar las pruebas, así logrando dividir los datos en 80 % para entrenamiento y 20 % para testeo del modelo. También se realizó la separación del archivo considerando cada clase para realizar las pruebas y entrenamiento del clasificador binario. Estos archivos se encuentran en la carpeta iris data del proyecto.

Entrenamiento del modelo

El entrenamiento del modelo se realizó con un 80 % de los datos que se encuentran en el archivo iris.csv. Luego se realizaron varios experimentos con diversos valores de alpha y para el número de neuronas de las capas ocultas. Los valores para alpha utilizados fueron: 2,0.5,0.1, 0.01, 0.001, 0.0001. Y se utilizaron los siguientes valores para el número de neuronas: [], [4], [5],[10],[20],[5,6],[1,5],[10,8]. Todos los experimentos consideraron un valor de epsilon igual a 0.0004 y 5000 iteraciones.

Resultados binarios

En el archivo experiment_results.csv se encuentran los datos obtenidos al experimentar con el clasificador binario para cada una de las clases. Podemos resumir los datos obtenidos en una tabla con los valores promedios para cada uno de los alphas:

Para la clase Iris-setosa:

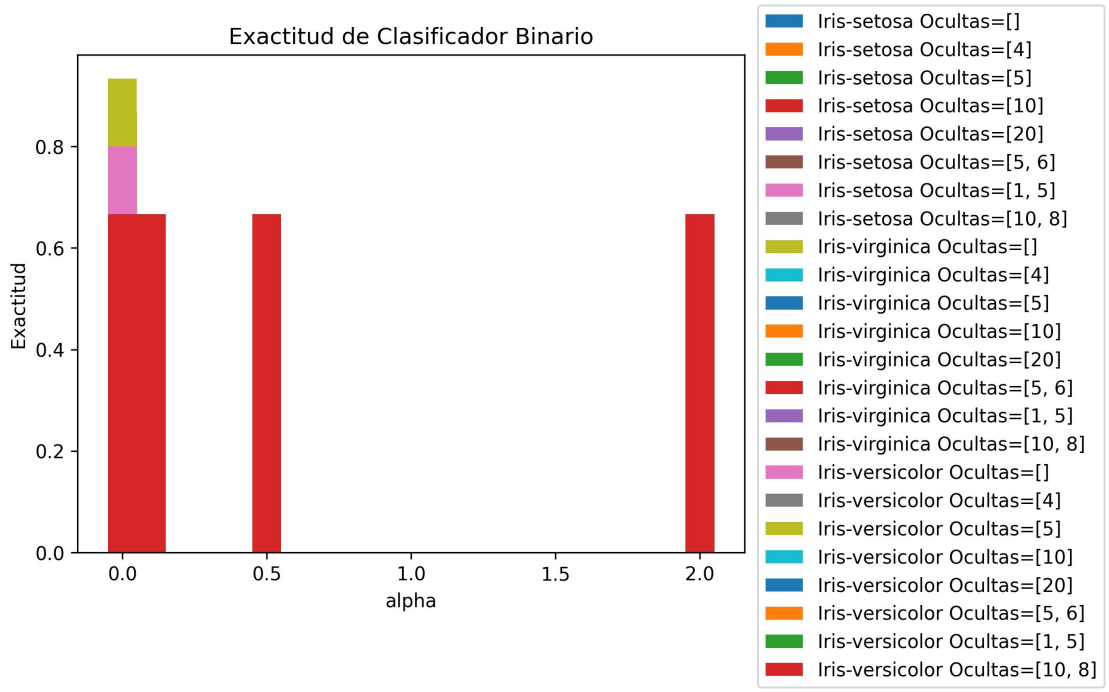
Para la clase Iris-virginica:

Para la clase Iris-versicolor:

clase	alpha	max error	min error	avg error	false positives	false negatives
Iris-setosa	2.0	0.9933435475468932	0.008066325293139375	0.3377738538927822	5.0	0.0
Iris-setosa	0.5	0.9901284783820289	0.012039979227709515	0.33950705210492543	5.0	0.0
Iris-setosa	0.1	0.9899489166225311	0.012420648064515363	0.3399395207093908	5.0	0.0
Iris-setosa	0.01	0.9882536929374193	0.013707593717136105	0.3401660384718401	5.0	0.0
Iris-setosa	0.001	0.9847716354442825	0.018377809625645752	0.3432939118246266	5.0	0.0
Iris-setosa	0.0001	0.9478732806218579	0.06809157931421397	0.3679522936250755	4.875	2.375

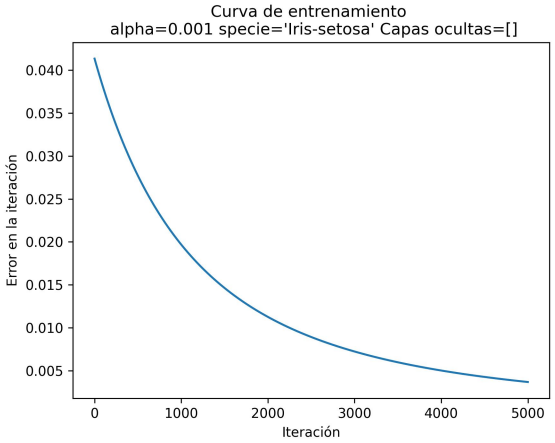
clase	alpha	max error	min error	avg error	false positives	false negatives
Iris-virginica	2.0	0.9711278000638742	0.007743792295801549	0.32907949250127516	4.625	0.0
Iris-virginica	0.5	0.9727461985458837	0.010315469762278465	0.3318042914032417	5.0	0.0
Iris-virginica	0.1	0.9782614733340316	0.011384733831399176	0.33430057636213717	5.0	0.0
Iris-virginica	0.01	0.9736006485910851	0.012020014443978	0.3332089868150744	5.0	0.0
Iris-virginica	0.001	0.9661494584161939	0.0139382733788313	0.33239072719663104	4.625	0.0
Iris-virginica	0.0001	0.9316080953506839	0.03261210877711486	0.3347701035122606	4.375	0.125

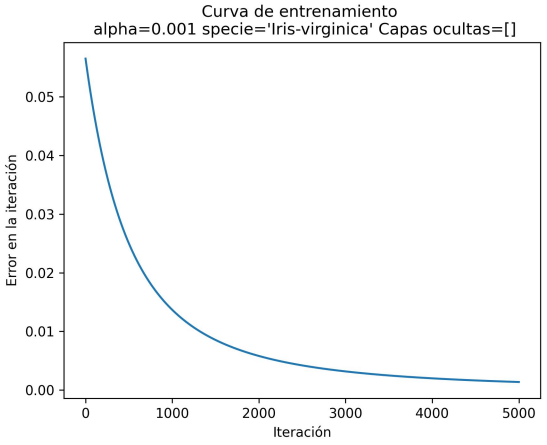
Tambien se calculó la exactitud del modelo obteniendo el siguiente gráfico para los experimentos con el clasificador binario.

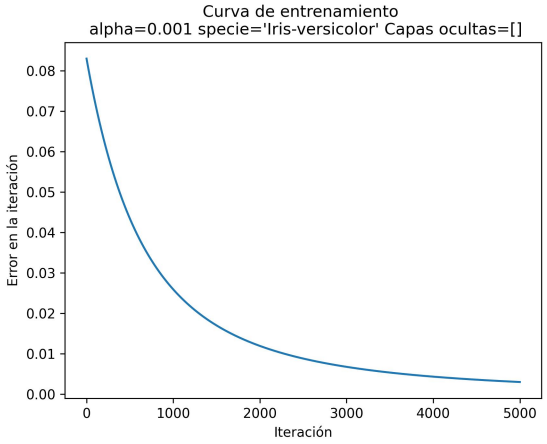


Se generaron gráficos de curva de entrenamiento que pueden ser encontrados en la carpeta plots del proyecto. En estos gráficos podemos observar que uno de los valores que logra mejorar la curva de entrenamiento es alpha 0.001. Podemos ver que para el caso lineal, sin ninguna capa oculta disminuye el error en la iteración al aumentar el número de iteraciones.

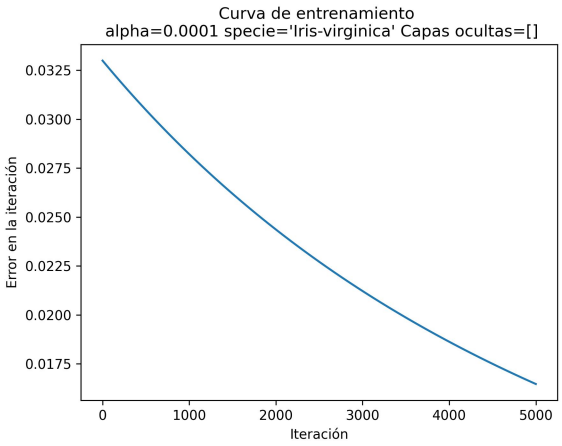
clase	alpha	max error	min error	avg error	false positives	false negatives
Iris-versicolor	2.0	0.9856648300814457	0.007766072663510938	0.33420140618809846	5.0	0.0
Iris-versicolor	0.5	0.9809487729938418	0.012018899094627625	0.33560431551894077	5.0	0.0
Iris-versicolor	0.1	0.9802994731551052	0.012555473250388038	0.3358329669061603	5.0	0.0
Iris-versicolor	0.01	0.9771213788286409	0.012803090462287201	0.3348514960393648	5.0	0.0
Iris-versicolor	0.001	0.9707221377878084	0.0151200784465387	0.3348606781367177	5.0	0.0
Iris-versicolor	0.0001	0.9279507023986	0.05042529147998087	0.3456814075992532	4.375	0.125

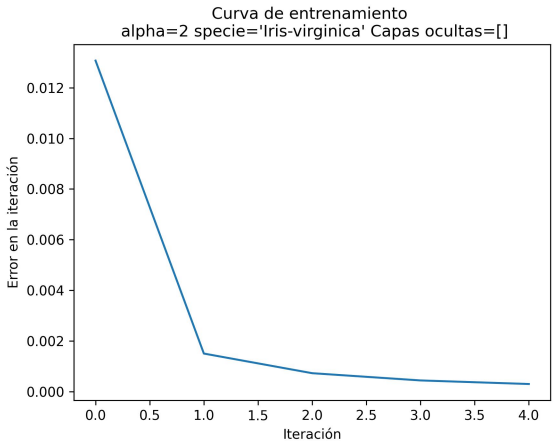






Podemos comparar estos gráficos a los obtenidos para otros valores como alpha igual a 2 y 0.0001.





Podemos ver que el error no disminuye rápidamente con el mismo número de iteraciones y además no disminuye de igual manera para cada una de las clases.
Por los resultados obtenidos con los experimentos del clasificador binario, el mejor valor de alpha es 0.001.

Además es posible corroborar que las clases son linealmente separables, debido a que

Resultados de multiclase

En los archivos experiment_results_multiclass_0.csv, experiment_results_multiclass_1.csv y experiment_results_multiclass_2.csv se encuentran los datos obtenidos al experimentar con el clasificador multiclase para cada una de las clases en su respectivo orden. Podemos resumir los datos obtenidos en una tabla con los valores promedios para cada uno de los alphas:

Para iris-setosa:

alpha	max error	min error	avg error	false positives	false negatives
2.0	0.002026741778523517	0.367181991029229	0.3258309314280196	0.0	0.0
0.5	0.007604411080846701	0.36320266140584645	0.319370520452945	0.0	0.0
0.1	0.06514019474641661	0.5495096409968006	0.40175653952953594	0.0	0.0
0.01	0.3736209306576805	0.6736765581758051	0.5805970641146649	0.0	5.125
0.001	0.6582535724790038	0.6670518216244958	0.648458953616241	0.0	10.0
0.0001	0.6834374873422551	0.655123836518507	0.7202497257227736	0.0	10.0

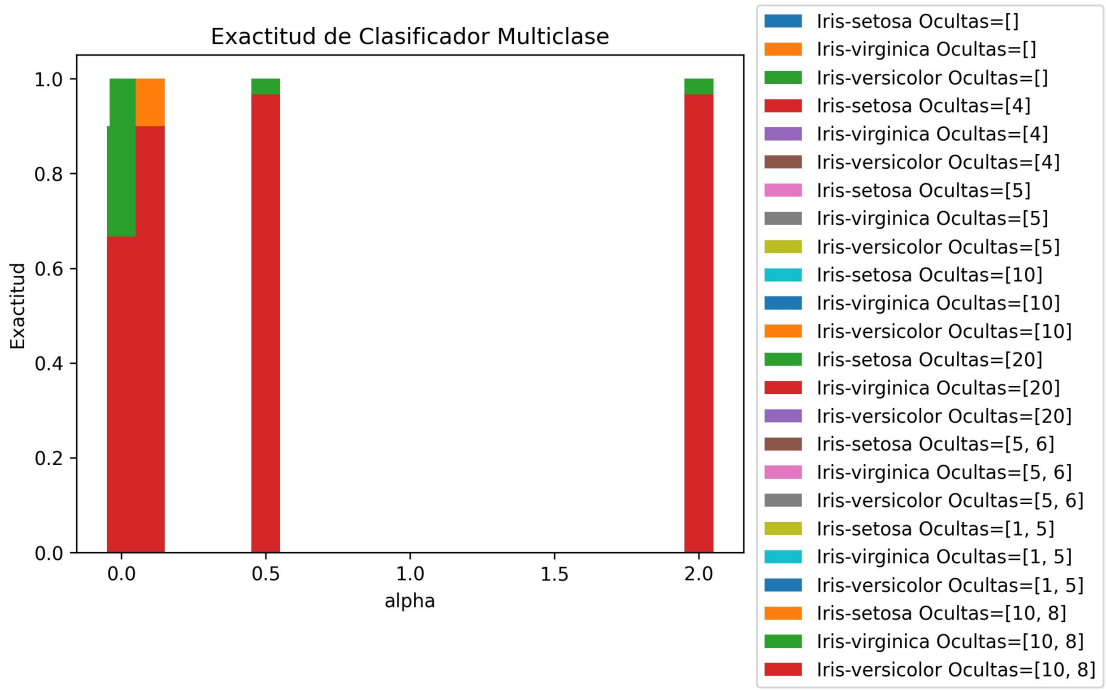
Para iris-virginica:

Para iris-versicolor:

Tambien se calculó la exactitud del modelo obteniendo el siguiente gráfico para los experimentos con el clasificador multiclase.

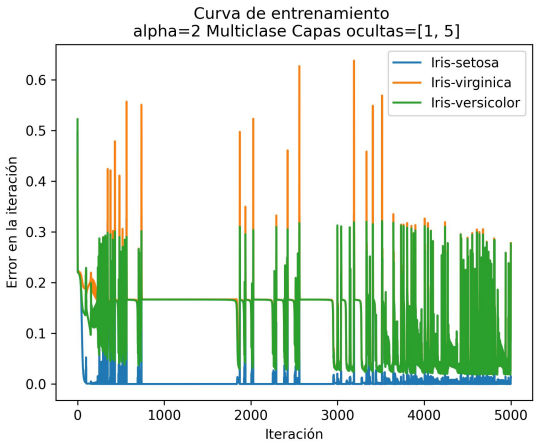
alpha	max error	min error	avg error	false positives	false negatives
2.0	8.597212942067507e-06	0.002081581234066479	1.5178501467954733e-09	0.0	2.5
0.5	8.703898939474 21e-06	0.002671874790620948	2.339192238411851e-06	0.0	1.25
0.1	1.8882063493900025e-05	0.023929621755791802	3.786790172270704e-05	0.0	3.5
0.01	0.1285182805999952	0.22737943361850316	0.14185941794620202	0.0	10.0
0.001	0.27488103267250297	0.32270590925218	0.3360900174881185	0.0	10.0
0.0001	0.39749828330285103	0.3868771136411261	0.3664751840738907	0.0	10.0

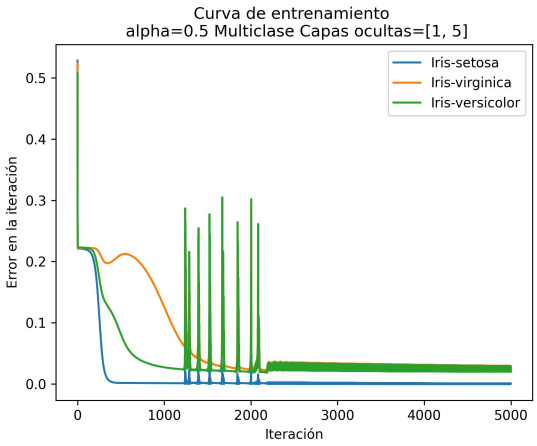
alpha	max error	min error	avg error	false positives	false negatives
2.0	0.0001827442890785733	0.0981336095147667	0.05144615016970099	0.0	2.125
0.5	0.0009384469672179125	0.05992605717490676	0.016098113161179987	0.0	0.875
0.1	0.00648588155628955	0.11452587726092721	0.04313374786529625	0.0	1.25
0.01	0.2121889550333064	0.4268148749934046	0.32229465398674106	0.0	10.0
0.001	0.4093428959526447	0.44502500195195654	0.44275846519172957	0.0	10.0
0.0001	0.5419842042415888	0.5284621206868994	0.5269400895035286	0.0	10.0



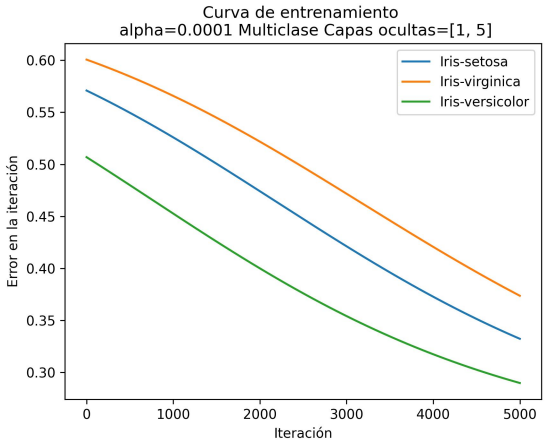
Por los resultados obtenidos con los experimentos del clasificador multiclase, el mejor valor de alpha es un alpha igual a 0.001.

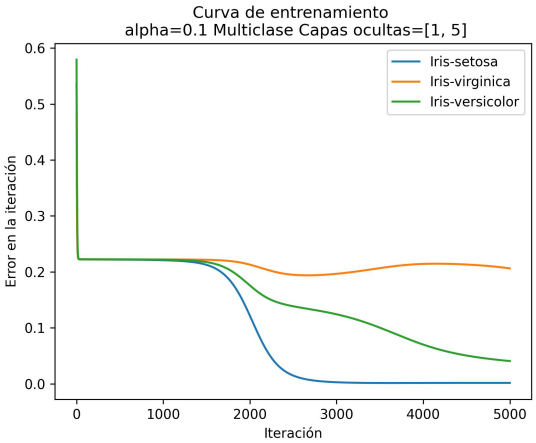
Se generaron gráficos de curva de entrenamiento que pueden ser encontrados en la carpeta plots del proyecto. Podemos ver por los gráficos que el único valor alpha que disminuye el error para todas las clases es 0.001.



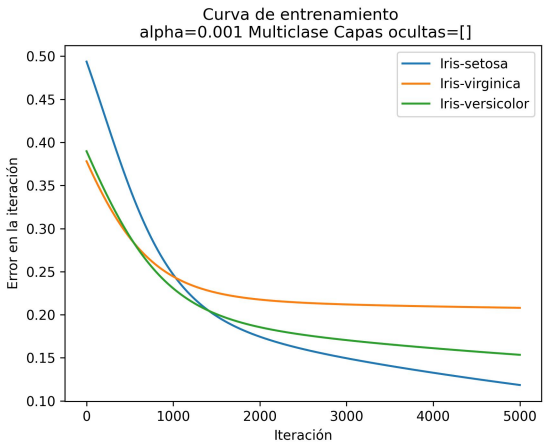


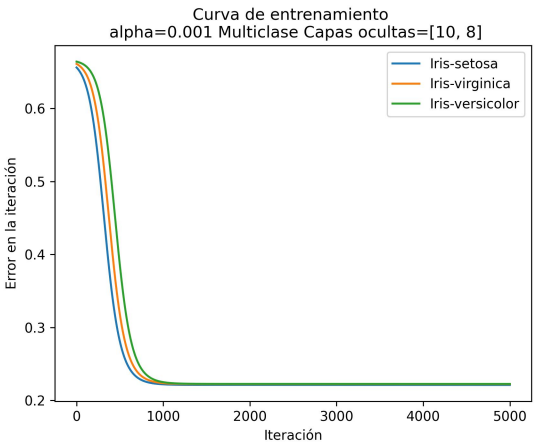
Para alpha 2 y 0.5 podemos observar que el clasificador multiclase presenta una gran cantidad de saltos en los errores de la iteración.

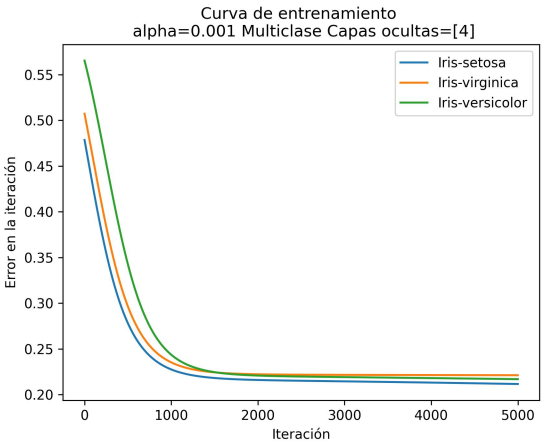


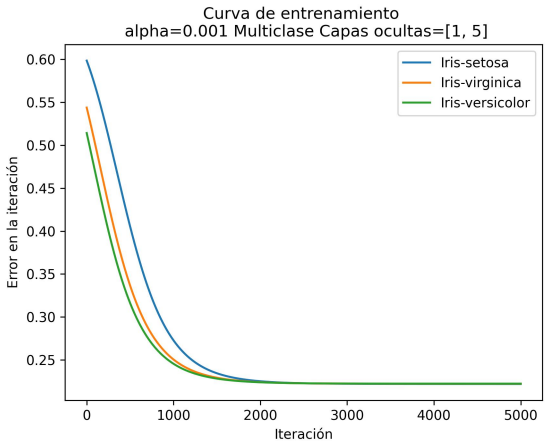


También para alpha 0.1 existen saltos en los errores y para 0.0001 no logra disminuir lo suficiente con 5000 iteraciones, a diferencia de alpha 0.001 como se muestra a continuación:









Que logra disminuir mucho más el error para todas las clases y con lo obtenido en el clasificador binario, podemos decir que alpha 0.001 será el valor escogido.

Comparación de los resultados

Por los datos obtenidos es posible observar que el clasificador multiclase fue capaz de disminuir la cantidad de errores que se obtienen para todas las clase, logrando también disminuir la cantidad de falsos positivos y falsos negativos.

Clasificador de Spam

Tratamiento de datos

El entrenamiento y prueba del clasificador binario de Spam es similar al realizado para las iris. Se normalizaron todas las columnas del dataset obteniendo que todos los valores se encuentren en el intervalo $[0, 1]$.

Division de datos

El dataset tiene un total de 4601 filas, de las cuales 1813 son Spam. El dataset se distribuyó de forma que los datos de pruebas están divididos en 1100 registros que son spam y 2480 que no lo son. De esta forma, se tiene una proporción del 69 % de correos que no son spam.

En el conjunto de prueba quedan un total de 1021 registros, de los cuales 713 de ellos son Spam, y, por tanto, 308 no lo son.

Esta division de los datos claramente favorece a la clase de los correos que no son Spam. Esto es intencional, pues, se quiere entrenar un clasificador que sea capaz de reconocer correos basura y a su vez no discrimine de forma agresiva, evitando que clasifique mal a correos que no son Spam. En esencia, se prefiere maximizar el número de falsos negativos en contra de los falsos positivos. Además, definiremos a un "buen clasificador de Spam" como aquel modelo que tenga la mayor cantidad de éxitos al clasificar el Spam pero a su vez tenga una baja cantidad de falsos positivos.

Entrenamiento y prueba

Se utilizaron 3000 iteraciones para hacer el entrenamiento y se establecieron dos métricas que varían. La tasa de aprendizaje toma uno de los siguientes valores $(2, 0{,}5, 0{,}1, 0{,}01, 0{,}001, 0{,}0001)$ y se usaron de 0 a 2 capas ocultas que toman 40 o 60 neuronas cada una. Esto da un total de 30 clasificadores diferentes entrenados y probados para conseguir la hipotesis posibles bajo estas condiciones. Cabe destacar que para que un clasificador considere a un correo electrónico como Spam, este debe tener una confianza del 80 %.

A continuación se mostrarán las tablas con los datos obtenidos fijando la topología de las

capas ocultas y variando la tasa de aprendizaje.

Primeramente, se ve la tabla de los clasificadores que no cuentan con capas ocultas. Estos es, todos los clasificadores tienen una capa de entrada de 56 neuronas y una capa de salida de una neurona.

alpha	test max error	test min error	test avg error	train max error	train min error	train avg error	false positives	false negatives
2.0000	0.999853	1.290698e-07	0.236721	0.998496	1.711426e-12	0.146622	7	296
0.5000	0.987428	8.478845e-05	0.297744	0.997262	1.391220e-07	0.192793	5	401
0.1000	0.944798	1.214048e-02	0.389261	0.973293	5.484542e-04	0.266051	3	589
0.0100	0.787369	2.000121e-01	0.515330	0.788786	1.475642e-01	0.380367	0	713
0.0010	0.663071	1.680770e-01	0.507099	0.971401	3.353973e-01	0.463960	0	712
0.0001	0.799749	7.124797e-02	0.400513	0.991837	1.143088e-01	0.562296	0	664

Puede verse que el clasificador con tasa de aprendizaje de 0.001 posee el menor error máximo de prueba. Sin embargo, tiene el segundo mayor error de prueba promedio. Por otra parte, el menor error de prueba promedio lo tiene el clasificador con tasa de aprendizaje 2.0. De hecho, en esta tabla, este es el clasificador que posee la mayor cantidad de aciertos al clasificar correctamente los correos Spam.

Ahora, veamos que ocurre con los clasificadores que contienen una capa oculta con 40 neuronas.

alpha	test max error	test min error	test avg error	train max error	train min error	train avg error	false positives	false negatives
2.0000	1.000000	8.846101e-11	0.163127	0.999934	1.482160e-20	0.094401	13	158
0.5000	1.000000	7.211488e-09	0.195121	0.999521	1.524860e-12	0.112960	8	218
0.1000	0.993729	9.691243e-04	0.298953	0.994824	1.189162e-04	0.190707	6	387
0.0100	0.738418	2.587056e-01	0.548797	0.733870	2.397937e-01	0.409531	0	713
0.0010	0.722991	2.732423e-01	0.574631	0.717847	2.584316e-01	0.427326	0	713
0.0001	0.998449	6.593631e-04	0.302509	0.999679	7.981809e-04	0.691016	308	0

En este caso, se puede ver a simple vista que, en promedio, el error promedio de prueba es menor en estos clasificadores comparados a los anteriores. Sin embargo, puede notarse que a medida que decrece al tasa de aprendizaje alpha, crece el error promedio de entrenamiento y de pruebas. Esto puede deberse al hecho de que el numero de iteraciones para el entrenamiento es fija. Por ello, el modelo no alcanza a ajustarse lo suficiente a los datos.

Vale la pena ver a los dos modelos que tienen la menor tasa de aprendizaje. Uno de ellos tiene un total de 713 falsos negativos. Este tipo de modelo es totalmente inútil, pues, para él ningun caso prueba es Spam. El mismo análisis puede realizarse sobre el último modelo. Este tiene 308 falsos positivos, esto significa que para él todo es Spam. Por tanto, ellos, y cualquier modelo parecido a ellos, son inútiles y muy malos clasificadores.

Vamos a centrarnos en los primeros tres clasificadores de esta tabla. Los tres cuentan con tasas de errores menores a 0.3 tanto para entrenamientos como para pruebas. Lo interesante de ellos son sus falsos negativos y falsos positivos. Se puede notar que el que menos falsos negativos tiene es el clasificador con el alpha igual a 2. Sin embargo, este clasificador también tiene un total de 13 falsos positivos. Por el contrario, el clasificador con alpha 0.1 tiene 6 y 387 falsos positivos y negativos, respectivamente. En teoria, los tres presentan buenas métricas para ser elegidos como un buen clasificador. Sin embargo, elegiremos al modelo cuyo alpha es 2. Pues, aunque él clasificó 5 falsos positivos más que el modelo con alpha 0.5, también es cierto que clasificó correctamente a 60 correos Spam más que los que hizo el otro. Esta comparación puede ser hecha de la misma forma con el modelo elegido de la tabla anterior. Y, de igual forma, el modelo con alpha 2 de esta tabla sigue siendo superior al de la anterior por el simple de hecho de tener más correos Spam clasificados correctamente.

A continuación, se muestra la tabla con los clasificadores que tienen una capa oculta con 60 neuronas:

alpha	test max error	test min error	test avg error	train max error	train min error	train avg error	false positives	false negatives
2.0000	1.000000	7.602980e-13	0.201574	0.999645	6.485235e-26	0.107135	7	260
0.5000	1.000000	8.206329e-09	0.194354	0.999486	1.127075e-13	0.111784	9	221
0.1000	0.985743	1.221320e-03	0.300745	0.988316	1.870669e-04	0.191607	7	396
0.0100	0.721887	2.452545e-01	0.554767	0.739941	2.377437e-01	0.413181	0	713
0.0010	0.738574	2.971084e-01	0.575615	0.735516	2.688209e-01	0.430746	0	713
0.0001	0.999712	8.741318e-05	0.301840	0.999960	1.213483e-04	0.692272	308	0

En esta tabla se han conseguido dos buenos clasificadores, los de los alpha 2.0 y 0.5. Ambos cuentan con buenas métricas en general. Pero vamos a fijarnos en sus falsos positivos y negativos. El del alpha 2 tiene tan solo 7 falsos positivos pero 260 falsos negativos. Por otra parte, el del alpha 0.5 tiene 9 falsos positivos y 221 falsos negativos. Ambos pueden catalogar como buenos clasificadores, pero es necesario elegir uno de ellos. En este caso, nos quedaremos con el del alpha 0.5. En este caso sacrificariamos 2 falsos positivos para ganar 39 correos de Spam correctamente clasificados. Sin embargo, al compararlo con el clasificador cuyo alpha es 2.0 y tiene una capa oculta con 40 neuronas, es preferible quedarse con el último. Sería de esperar que al aumentar el número de neuronas, también se aumente la calidad del clasificador. Sin embargo, esto no es cierto.

Veamos que ocurre cuando se agrega una nueva capa oculta, los siguientes clasificadores cuentan con dos capas ocultas y 40 neuronas en cada una:

alpha	test max error	test min error	test avg error	train max error	train min error	train avg error	false positives	false negatives
2.0000	1.000000	1.834454e-08	0.301665	1.000000	7.112244e-15	0.307263	308	0
0.5000	1.000000	9.582490e-11	0.698335	0.934897	6.510262e-02	0.667642	0	713
0.1000	0.692738	3.072615e-01	0.576453	0.692738	3.072615e-01	0.425705	0	713
0.0100	0.692737	3.072625e-01	0.576453	0.692737	3.072625e-01	0.425705	0	713
0.0010	0.692738	3.072623e-01	0.576453	0.692738	3.072621e-01	0.425705	0	713
0.0001	0.939919	6.008049e-02	0.325497	0.940212	5.978807e-02	0.669690	308	0

De inmediato, es posible ver que todos estos clasificadores son inútiles, ninguna cuenta con métricas siquiera decentes para poder considerarse como válido.

Aumentemos la cantidad de neuronas en cada capa para ver los cambios

alpha	test max error	test min error	test avg error	train max error	train min error	train avg error	false positives	false negatives
2.0000	1.000000	0.000000e+00	0.301665	1.000000	1.845704e-14	0.307263	308	0
0.5000	1.000000	7.289643e-08	0.698335	0.582110	4.178904e-01	0.531651	0	713
0.1000	0.944502	5.549780e-02	0.676321	0.559027	4.409727e-01	0.522754	0	713
0.0100	0.692737	3.072626e-01	0.576453	0.692737	3.072626e-01	0.425705	0	713
0.0010	0.692737	3.072626e-01	0.576453	0.692737	3.072626e-01	0.425705	0	713
0.0001	0.997639	2.361022e-03	0.302602	0.997659	2.341266e-03	0.691835	308	0

De igual forma que el caso anterior, sólo se han obtenido clasificadores inútiles. Por tanto, no se obtiene ningún beneficio si se siguen aumentando las capas o las neuronas.

Así, la hipotesis que elegimos como el mejor clasificador es la que se obtiene al entrenar al modelo con 3000 iteraciones, un alpha de 2 y una capa oculta con 40 neuronas.