Activación y optimización de redes neuronales

Mathematical Engineering - Ingenieria Matematica

Autores: Camilo Oberndorfer, Miguel Valencia, M. Alejandra Moncada, Pedro Botero

Asesor: Prof. Cristhian David Montoya Zambrano

Semestre académico: 2021-2

1. Introducción

El uso de redes neuronales es una importante herramienta para resolver problemas de clasificación y reconocimiento hoy en dia, por lo cual es importante buscar la forma de minimizar el error cometido por estas al momento de dar un resultado pero al mismo tiempo agilizar el procesamiento de los datos y la obtención de resultados por medio de distintas formas de aprendizaje y distintas funciones de activación.

Se propondrá un perceptrón para la solución de múltiples datasets de distintas formas para ver su comportamiento y exactitud en cada caso según sus características específicas y de esta manera ver las ventajas y desventajas de usar cada uno de los métodos de optimización y funciones de activación.

2. Planteamiento del Problema

Una red neuronal consta de una o varias capas de neuronas conectadas mediante enlaces. Cada neurona calcula un valor de salida a partir de una combinación lineal de los valores de entrada de la capa previa de la forma

$$\sum_{i=1}^{n} w_i x_i + B$$

donde x_i se refiere al dato de entrada i, w_i se refiere al peso asignado en cada neurona al dato de entrada i y B se refiere al umbral o desface asignado a cada neurona.

De esta forma, el valor dado por cada neurona está dado por la anterior combinación lineal usada dentro de una función de activación f, por lo cual el output de cada neurona está calculado de la forma.

$$y_s = f(\sum_{i=1}^{n} w_i x_i + B) + \epsilon_s$$

Donde y_s denota el output de la neurona s y ϵ_s es el error asociado al calculo del output de la neurona s.

El objetivo del trabajo es plantear las posibles maneras de minimizar el error cometido por la red neuronal. Este proceso de aprendizaje para minimizar el error cometido está dado de la forma: minimizar ϵ sujeto a

$$w_0 + \sum_{i=1}^{I} w_i x_i s - \epsilon \leq f^{-1}(y_s), s = 1, ..., S.$$

3. Teoría

Una red neuronal es un conjunto de elementos, unidades o nodos, los cuales están basados en el funcionamiento de una red neuronal animal. La habilidad de procesamiento de la red está almacenada en los pesos, obtenidos en un proceso de adaptación , o de aprendizaje obtenido de un set de entrenamiento.
En el aprendizaje de la red neuronal para una mejora de los resultados se tienen varias propuestas por métodos de optimización y de aprendizaje general, el método mas fácil de implementar es el de escoger sesgos y pesos

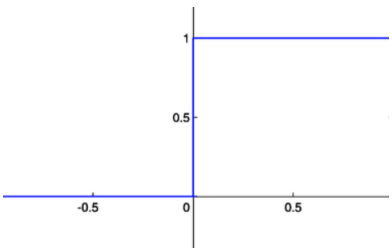
aleatorios dentro de los valores del output de cada neurona, este método es muy poco efectivo ya que no tiene una dirección establecida que intenté disminuir el error del resultado con cada paso que da y además trae el problema de no asegurar dar un resultado basado en un mínimo local o global de los parámetros dados por los pesos y el sesgo a diferencia de otros métodos de aprendizaje basados en los métodos de optimización no lineal L-BFGS, Gradiente descendiente, gradiente descendiente estocástico y gradiente conjugado no lineal.

Con los métodos anteriormente mencionados se crea un proceso de aprendizaje para el perceptrón que está basado en las derivadas parciales de la función de pérdida de los outputs finales de predicción dados por la red neuronal, esto crea un proceso de aprendizaje donde la modificación de los pesos esta direccionada hacia por lo menos un mínimo local por lo que asegura casi por completo que habrá una mejora en las predicciones con cada Batch de entrenamiento.

3.1. Funciones de activación

3.1.1 Step

La función de activación step establece que los valores de los nodos que son menores o iguales a cero toman el valor de cero y los valores mayores a cero representan el número uno, de esta manera funcionando como una forma de conversión a booleano que facilita un resultado final de este tipo, sin embargo no es la mejor opción ya que tiene grandes problemas al representar y predecir datasets que no tengan una separación lineal clara.

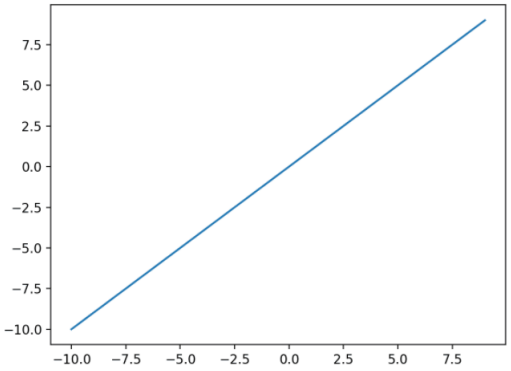


3.1.2 Lineal

La función de activación lineal simplemente toma los valores que da la operación inicial de la neurona

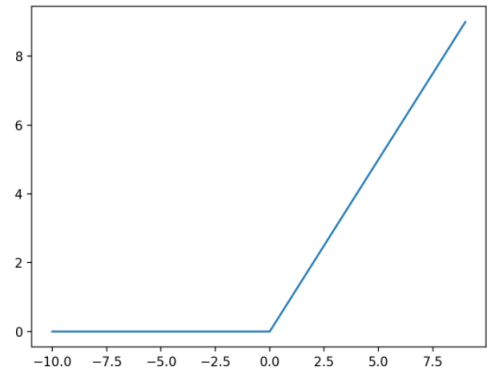
$$o = \sum_{j=1}^{n} w_n i_n + b$$

y no los modifica, esto quiere decir que todas las representaciones por medio de esta función de activación son lineales por lo cual esta no es una buena opción para predecir datasets que no son linealmente separables.



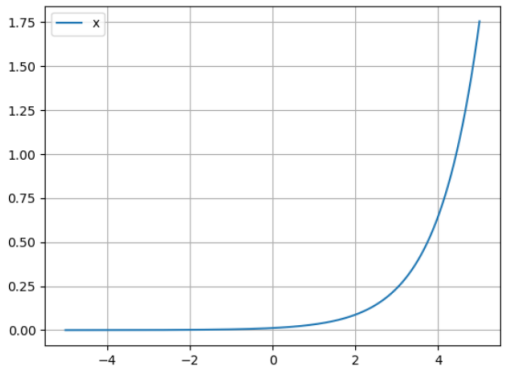
3.1.3 ReLU

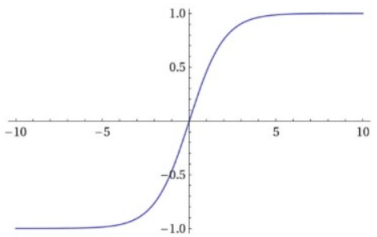
La función rectified linear unit o ReLU utiliza una combinación entre la función step y la función lineal de manera que cuando los valores del output iniciales son menores a 0, la función toma el valor de cero y cuando toma valores mayores a cero, estos toman valores de forma lineal, de esta manera se pueden aproximar datasets no linealmente separables por la posibilidad de activar ciertos nodos mientras otros permanecen inactivos, esta es de las funciones de activación mas utilizadas por su facilidad para la impementación, su versatilidad al momento de representar datasets extraños y su gran mejora al incluir una poca cantidad de capas ocultas de nodos.



3.1.4 Softmax

La función softmax se basa en la función exponencial e^x para todos los valores negativos, de esta manera todos los valores tienen una representación entre cero y uno, así se soluciona el mayor problema que tiene la activación ReLU que es que cuando el output de un nodo es negativo la información que este da se pierde dentro de la función de activación, sin embargo para evadir que los valores positivos mayores a uno tomen un valor exponencial muy alto, los datos se estandarizan restándole a todos los inputs el mayor valor de los inputs, de esta manera todos serán negativos y tendrán una representación del valor entre cero y uno.





3.2. Optimizadores

3.2.1 Gradiente Descendiente estocástico

Se utiliza hallando las derivadas parciales de la función de perdida con respecto a los pesos, outputs y sesgos de los nodos anteriores, y con estos los anteriores hasta llegar a la capa de entradas, de esta manera se restan los gradientes a cada peso de cada nodo de cada capa y así con respecto al resultado del batch de datos inspeccionados busca un mínimo local.

Mas específicamente lo que hace el método es establecer un grado de aprendizaje inicial entre cero y uno, hacer el proceso de hallar la derivada de los pesos, inputs, outputs y sesgos y cambiar el valor de todos los pesos y sesgos por un factor del grado de aprendizaje por el gradiente del peso o sesgo especifico, esto quiere decir que el grado de aprendizaje funciona como un tamaño de paso y el gradiente como la dirección hacia donde ir para minimizar el error.

Con respecto van pasando las iteraciones de aprendizaje con los Batches establecidos se va disminuyendo el tamaño de paso o grado de aprendizaje por un factor de un parámetro inicial nombrado gasto de aprendizaje por el numero de la iteración en la que nos encontramos, de esta manera calcular el grado de aprendizaje de cada iteración esta dado por la ecuación:

$$ga(d,s) = ga(0,0)\frac{1}{1+(d*s)}$$

Donde ga es grado de aprendizaje, s es el numero de la iteración actual y d es la taza de gasto de aprendizaje. Eso se hace con el objetivo de que sea mucho mas difícil que nuestra aproximación se pase del valor mínimo esperado.

Por último opcionalmente se puede añadir momentum al método de optimización de forma que si nuestros valores se están acercando a un mínimo local sea mas probable que pasen por este y sigan buscando el mínimo global que mejor aproxime los resultados, esto se hace usando un parámetro inicial entre 0 y 1 llamado taza de momentum que luego se multiplica por los cambios anteriores a los pesos, de esta manera obtenemos que la forma mas completa de calcular el cambio a hacer a los pesos esta dada por la ecuación:

$$\nabla w_i = (\rho \nabla w_{i-1}) - (ga_i O'_w)$$

Donde w_i es el peso en la iteración i, ρ es la

tasa de momentum, ga_i es la tasa de aprendizaje de la iteración i y O'_w es el gradiente del peso encontrado. Se tiene entonces que los pesos se calculan de la forma:

$$w_i = w_{i-1} + \nabla w_i$$

3.2.2 AdaGrad

Uno de los problemas principales que se encuentran al trabajar con redes neuronales es la elección previa de la tasa de aprendizaje. Ya que una mala elección de esta misma puede llevar a que el algoritmo se demore gran cantidad de tiempo en llegar a una pérdida que se considere aceptable en caso de escogerlo muy pequeño, o, directamente nunca llegar a un punto en el que se acepte la pérdida si se le da un valor muy grande al parámetro. AdaGrad busca precisamente solucionar esto asignando tasas de aprendizaje únicas por parámetro en vez de una tasa global para toda la red neuronal.

AdaGrad, también conocido como Adaptive Gradient Algorithm, busca normalizar las actualizaciones que se dan durante el periodo de aprendizaje, generando así que la diferencia entre pesos nunca sea demasiado grande. Esto lo logra al almacenar un registro de las actualizaciones previas, donde mientras más grande sea la suma de las actualizaciones, menor será la cantidad de actualizaciones que se realicen durante el periodo de aprendizaje. Se establecen entonces las siguientes fórmulas

$$\text{cache} \mathrel{+}= \text{gradiente}^2$$

$$\text{actualizaciones} \ = \eta \cdot \frac{gradiente}{\sqrt{cache} + eps}$$

El cache es la sumatoria del histórico del cuadrado de los gradientes. La división que se presenta a medida que el cache va aumentando, genera que las actualizaciones sean cada vez menores causando así que el aprendizaje sea cada vez menor a medida que pasa el tiempo. Por esta razón, este optimizador no es comúnmente utilizado a excepcion de algunas aplicaciones específicas. Epsilon, es un hiper-parámetro que impide que sea realice una división por 0.

En general, AdaGrad genera que las tasas de aprendizaje para los parámetros con gradientes más pequeños disminuyen lentamente, mientras que aquellas para los parámetros con gradientes más grandes decrecen más rápido. Llegamos entonces a que la fórmula para realizar el cálculo de los pesos en la siguiente iteración es igual a:

$$w_i = w_{i-1} - \frac{ga_i \cdot O'_w}{\sqrt{C_{i-1,w}} + \epsilon}$$

A su vez, el cálculo del sesgo en la siguiente iteración es igaul a:

$$b_i = b_{i-1} - \frac{ga_i \cdot O'_b}{\sqrt{C_{i-1,b}} + \epsilon}$$

Finalmente, el cálculo para identificar la siguiente tasa de aprendizaje se puede representar con la fórmula:

$$ga_i = ga_{i-1} \cdot \frac{1}{1 + d \cdot iter}$$

3.2.3 RMSprop

Root Means Square Propagation es una extensión del método del gradiente descendiente estocástico. Este método está diseñado para acelerar el proceso de optimización,ya sea reduciendo el número de evaluaciones necesarias o mejorando la capacidad del algoritmo obteniendo un mejor resultado.
Al igual que AdaGrad asigna tasas de aprendizaje únicas por parámetro, pero las calcula de una forma diferente. RMSprop usa un promedio variante de las tasas de aprendizaje que permite que durante el proceso de búsqueda el algoritmo olvide los gradientes anteriores y se centre en los más recientes. Lo anterior evita que se disminuya tasa de aprendizaje de manera monótona y se ralentice el proceso de búsqueda como ocurre en el AdaGrad.
La formula media derivada parcial cuadrática para un parámetro esta dada:

$$s(t+1) = (s(t)\rho) + (f'(x(t))^2(1-\rho))$$

Donde $s(t+1)$ y $s(t)$ son el cache de la iteración actual y la iteración anterior respectivamnete, $f'(x(t))^2$ es el gradiente elevado al cuadrado, y ρ es un hiperparámetro que estaremos trabajando como el momentum. Dado lo anterior el tamaño paso ajustado puede verse así:

$CStepSize(t+1) = stepSize/(1e{-}8{+}RMS(s(t+1)))$

Una vez obtengamos el tamaño paso ajustado para el parámetro , podemos actualizar el parámetro utilizando tamaño paso ajustado y el gradiente.

$$x(t+1) = x(t) - CStepSize(t+1)f'(x(t))$$

Este proceso se repite para variable de entrada hasta que un nuevo punto en el espacio de búsqueda es creado y pueda ser evaluado.

3.2.4 Adam

El nombre Adam viene de adaptative momentum por lo que se basa en el uso de momentums en vez de derivadas como en el método de gradiente descendiente estocástico con momentum y en el uso de cambios de pesos adaptativos como el visto en RMSprop, este es el método de optimización mas usado para el aprendizaje de redes neuronales ya que ha mostrado llegar a una mayor precisión al predecir resultados y tener una pérdida menor, también ha mostrado ser mejor trabajando con datasets complicados.

Este método tiene un mecanismo de corrección para el sesgo en el cual se divide a tanto el cache como el momentum por un valor $1-\beta_j^i$ para la iteración i, el valor β_j inicial es un valor entre 0 y 1 y el j denota si este es usado para el momentum o para el cache, de forma que se pueden usar dos valores de β distintos.

Por esto los cambios en los pesos de los primeros pasos no serán tan afectados por el periodo de calentamiento en el cual la red neuronal tiene unos pesos lejanos a los que se buscan, esto es porque el cache tendrá inicialmente valores muy bajos por las pocas iteraciones y el momentum será bajo o cero ya que no hay cambios en los pesos anteriores o son muy bajos estos cambios, el hecho de dividir por un numero cercano a 0 de la forma:

$$\frac{cache}{1-(\beta_1)^i}$$

$$\frac{momentum}{1-(\beta_2)^i}$$

aumentará el tamaño de paso inicialmente pero despues de varias iteraciones no lo afectará de gran manera.

El uso del momentum y el cache sugiere la siguiente formula para calcular los pesos de la siguiente iteración:

$$w_i = w_{i-1} - \frac{(ga_i(\frac{\beta_1\rho_{i-1}+(1-\beta_1)O'_w}{(1-\beta_1^i)}))}{\sqrt{\frac{\beta_2 C_{i-1}+(1-\beta_2)(O'_w)^2}{(1-\beta_2^i)}}+\epsilon}$$

y para el calculo del siguiente sesgo:

$$b_i = b_{i-1} - \frac{(ga_i(\frac{\beta_1\rho_{i-1}+(1-\beta_1)O'_b}{(1-\beta_1^i)}))}{\sqrt{\frac{\beta_2 C_{i-1}+(1-\beta_2)(O'_b)^2}{(1-\beta_2^i)}}+\epsilon}$$

4. Herramientas computacionales

Se usa el lenguaje Python versión 3.7 con las siguientes librerias y por las siguientes razones:

1. Numpy: se utiliza para las operaciones mariciales de producto punto y matriz transversa, para organizar los datos de una manera mas accesible e intuitiva.

2. Pandas: se usa para imprimir los datos (inputs, outputs y pesos) de manera organizada.

3. nnfs: ayuda a crear datasets de flotantes de distintas formas que hacen fácil probar la red neuronal con casos extremos y distintos. (si tienes un dataset previo no es necesario importarlo).

4. matplotlib: permite graficar los datasets creados para visualizarlos mas claramente y ver la clasificación de la red neuronal dependiendo del método de optimización y función de activación utilizados.

De las librerias anteriormente mencionadas solo se utilizan funciones básicas que no están sujetas a cambios y actualizaciones constantes por lo cual es muy posible que no afecte la versión utilizada.

5. Resolución del problema

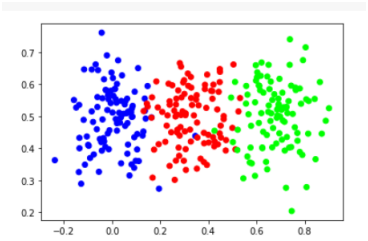
Se usaron dos distintos datasets para el entrenamiento y test de los perceptrones con los

distintos optimizers y funciones de activación y se hallaron los resultados de precisión al predecir por cada una de las redes neuronales construidas.

5.1. Datasets

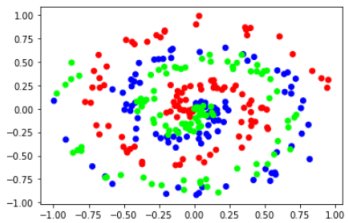
5.1.1 Separación lineal

Este dataset se basa en datos separados en 3 grupos distintos que por su distribución serian linealmente separables.



5.1.2 Espiral

Este dataset consta de tres grupos que no constan con una separación lineal entre ellos por lo cual predecir a que grupo pertenece un individuo es considerablemente mas difícil que en el caso anterior.



5.2. Optimizadores

Aquí veremos el resultado en 10000 epochs o pasos de tiempo (iteraciones) de cada uno de los optimizadores con una red neuronal de una capa oculta donde la función de activación para la primera capa es ReLU, lineal o de paso y la segunda capa utiliza siempre la función softmax con el propósito de mantener los valores resultantes entre 0 y 1 y sin perder información.

Para comprender las tablas que se presentan a continuación, es necesario tener en cuenta que epoch, se refiere al número de iteración o paso; acc explica la precisión de 0 a 1, donde a mayor número mayor es la precisión; loss, es la pérdida que se presenta en eso paso de la red neuronal y finalmente, lr es la tasa de aprendizaje que se tiene en dicho paso de la red.

5.2.1 Random

- Utilizando el dataset de separación lineal:
  - Lineal

```
epoch: 0, acc: 0.333, loss: 1.099,
epoch: 1000, acc: 0.337, loss: 10.419,
epoch: 2000, acc: 0.333, loss: 10.745,
epoch: 3000, acc: 0.333, loss: 10.745,
epoch: 4000, acc: 0.333, loss: 10.745,
epoch: 5000, acc: 0.333, loss: 10.745,
epoch: 6000, acc: 0.333, loss: 10.745,
epoch: 7000, acc: 0.333, loss: 10.745,
epoch: 8000, acc: 0.597, loss: 5.569,
epoch: 9000, acc: 0.353, loss: 9.250,
epoch: 10000, acc: 0.333, loss: 10.745,
```

Figure 1: Optimización Random con función de activación lineal

  - Step

```
epoch: 0, acc: 0.007, loss: 1.114,
epoch: 1000, acc: 0.333, loss: 6.010,
epoch: 2000, acc: 0.333, loss: 10.745,
epoch: 3000, acc: 0.333, loss: 10.745,
epoch: 4000, acc: 0.333, loss: 10.745,
epoch: 5000, acc: 0.333, loss: 10.745,
epoch: 6000, acc: 0.333, loss: 10.745,
epoch: 7000, acc: 0.333, loss: 10.745,
epoch: 8000, acc: 0.333, loss: 10.745,
epoch: 9000, acc: 0.333, loss: 10.745,
epoch: 10000, acc: 0.333, loss: 10.745,
```

Figure 2: Optimización Random con función de activación step

  - ReLU

```
epoch: 0, acc: 0.333, loss: 1.099,
epoch: 1000, acc: 0.333, loss: 10.745,
epoch: 2000, acc: 0.333, loss: 10.745,
epoch: 3000, acc: 0.333, loss: 10.745,
epoch: 4000, acc: 0.333, loss: 10.745,
epoch: 5000, acc: 0.333, loss: 10.745,
epoch: 6000, acc: 0.333, loss: 10.745,
epoch: 7000, acc: 0.333, loss: 10.745,
epoch: 8000, acc: 0.333, loss: 10.745,
epoch: 9000, acc: 0.333, loss: 10.745,
epoch: 10000, acc: 0.333, loss: 10.745,
```

Figure 3: Optimización Random con función de activación ReLU

- Utilizando el dataset del espiral:
  - Lineal

```
epoch: 0, acc: 0.333, loss: 1.099,
epoch: 1000, acc: 0.373, loss: 9.553,
epoch: 2000, acc: 0.287, loss: 10.051,
epoch: 3000, acc: 0.287, loss: 10.723,
epoch: 4000, acc: 0.343, loss: 9.734,
epoch: 5000, acc: 0.323, loss: 10.569,
epoch: 6000, acc: 0.317, loss: 10.896,
epoch: 7000, acc: 0.313, loss: 10.989,
epoch: 8000, acc: 0.337, loss: 10.692,
epoch: 9000, acc: 0.333, loss: 10.745,
epoch: 10000, acc: 0.343, loss: 10.584,
```

Figure 4: Optimización Random con función de activación lineal

- Step

```
epoch: 0, acc: 0.330, loss: 1.096,
epoch: 1000, acc: 0.333, loss: 10.745,
epoch: 2000, acc: 0.333, loss: 10.745,
epoch: 3000, acc: 0.333, loss: 10.745,
epoch: 4000, acc: 0.333, loss: 10.745,
epoch: 5000, acc: 0.333, loss: 10.745,
epoch: 6000, acc: 0.333, loss: 10.745,
epoch: 7000, acc: 0.333, loss: 10.745,
epoch: 8000, acc: 0.333, loss: 10.745,
epoch: 9000, acc: 0.333, loss: 8.304,
epoch: 10000, acc: 0.333, loss: 10.745,
```

Figure 5: Optimización Random con función de activación step

- ReLU

```
epoch: 0, acc: 0.333, loss: 1.099,
epoch: 1000, acc: 0.333, loss: 10.745,
epoch: 2000, acc: 0.333, loss: 10.745,
epoch: 3000, acc: 0.333, loss: 10.745,
epoch: 4000, acc: 0.333, loss: 10.745,
epoch: 5000, acc: 0.333, loss: 10.745,
epoch: 6000, acc: 0.333, loss: 10.745,
epoch: 7000, acc: 0.333, loss: 10.745,
epoch: 8000, acc: 0.333, loss: 10.745,
epoch: 9000, acc: 0.333, loss: 10.745,
epoch: 10000, acc: 0.333, loss: 10.745,
```

Figure 6: Optimización Random con función de activación ReLU

Además, se plantea el siguiente gráfico que representa la pérdida usando el dataset del espiral con la función ReLU:

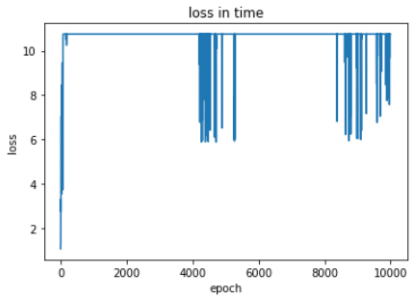


Figure 7: Pérdida del espiral con función ReLU

5.2.2 AdaGrad

- Utilizando el dataset de separación lineal:
  - Lineal

```
epoch: 0, acc: 0.333, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.953, loss: 0.112, lr: 0.049975037468784345
epoch: 2000, acc: 0.953, loss: 0.112, lr: 0.04995007490013731
epoch: 3000, acc: 0.953, loss: 0.112, lr: 0.049925137256683606
epoch: 4000, acc: 0.953, loss: 0.112, lr: 0.049900224501110035
epoch: 5000, acc: 0.953, loss: 0.112, lr: 0.04987533659617785
epoch: 6000, acc: 0.953, loss: 0.112, lr: 0.04985047350472258
epoch: 7000, acc: 0.953, loss: 0.112, lr: 0.04982563518965381
epoch: 8000, acc: 0.953, loss: 0.112, lr: 0.04980082161395499
epoch: 9000, acc: 0.953, loss: 0.112, lr: 0.04977603274068329
epoch: 10000, acc: 0.953, loss: 0.112, lr: 0.04975126853296942
```

Figure 8: Optimización AdaGrad con función de activación lineal

- Step

```
epoch: 0, acc: 0.413, loss: 1.076, lr: 0.05
epoch: 1000, acc: 0.957, loss: 0.127, lr: 0.049975037468784345
epoch: 2000, acc: 0.957, loss: 0.118, lr: 0.04995007490013731
epoch: 3000, acc: 0.957, loss: 0.115, lr: 0.049925137256683606
epoch: 4000, acc: 0.957, loss: 0.113, lr: 0.049900224501110035
epoch: 5000, acc: 0.957, loss: 0.111, lr: 0.04987533659617785
epoch: 6000, acc: 0.957, loss: 0.110, lr: 0.04985047350472258
epoch: 7000, acc: 0.957, loss: 0.110, lr: 0.04982563518965381
epoch: 8000, acc: 0.957, loss: 0.109, lr: 0.04980082161395499
epoch: 9000, acc: 0.957, loss: 0.108, lr: 0.04977603274068329
epoch: 10000, acc: 0.957, loss: 0.108, lr: 0.04975126853296942
```

Figure 9: Optimización AdaGrad con función de activación step

- ReLU

```
epoch: 0, acc: 0.333, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.957, loss: 0.113, lr: 0.049975037468784345
epoch: 2000, acc: 0.953, loss: 0.112, lr: 0.04995007490013731
epoch: 3000, acc: 0.953, loss: 0.112, lr: 0.049925137256683606
epoch: 4000, acc: 0.953, loss: 0.112, lr: 0.049900224501110035
epoch: 5000, acc: 0.953, loss: 0.112, lr: 0.04987533659617785
epoch: 6000, acc: 0.953, loss: 0.112, lr: 0.04985047350472258
epoch: 7000, acc: 0.953, loss: 0.112, lr: 0.04982563518965381
epoch: 8000, acc: 0.953, loss: 0.112, lr: 0.04980082161395499
epoch: 9000, acc: 0.953, loss: 0.112, lr: 0.04977603274068329
epoch: 10000, acc: 0.953, loss: 0.112, lr: 0.04975126853296942
```

Figure 10: Optimización AdaGrad con función de activación ReLU

- Utilizando el dataset del espiral:
  - Lineal

```
epoch: 0, acc: 0.333, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.417, loss: 1.071, lr: 0.049975037468784345
epoch: 2000, acc: 0.417, loss: 1.071, lr: 0.04995007490013731
epoch: 3000, acc: 0.417, loss: 1.071, lr: 0.049925137256683606
epoch: 4000, acc: 0.417, loss: 1.071, lr: 0.049900224501110035
epoch: 5000, acc: 0.417, loss: 1.071, lr: 0.04987533659617785
epoch: 6000, acc: 0.417, loss: 1.071, lr: 0.04985047350472258
epoch: 7000, acc: 0.417, loss: 1.071, lr: 0.04982563518965381
epoch: 8000, acc: 0.417, loss: 1.071, lr: 0.04980082161395499
epoch: 9000, acc: 0.417, loss: 1.071, lr: 0.04977603274068329
epoch: 10000, acc: 0.417, loss: 1.071, lr: 0.04975126853296942
```

Figure 11: Optimización AdaGrad con función de activación lineal

- Step

```
epoch: 0, acc: 0.337, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.703, loss: 0.693, lr: 0.049975037468784345
epoch: 2000, acc: 0.717, loss: 0.658, lr: 0.04995007490013731
epoch: 3000, acc: 0.720, loss: 0.641, lr: 0.049925137256683606
epoch: 4000, acc: 0.720, loss: 0.631, lr: 0.049900224501110035
epoch: 5000, acc: 0.720, loss: 0.625, lr: 0.04987533659617785
epoch: 6000, acc: 0.723, loss: 0.620, lr: 0.04985047350472258
epoch: 7000, acc: 0.723, loss: 0.617, lr: 0.04982563518965381
epoch: 8000, acc: 0.723, loss: 0.614, lr: 0.04980082161395499
epoch: 9000, acc: 0.730, loss: 0.612, lr: 0.04977603274068329
epoch: 10000, acc: 0.733, loss: 0.610, lr: 0.04975126853296942
```

Figure 12: Optimización AdaGrad con función de activación step

- ReLU

```
epoch: 0, acc: 0.333, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.417, loss: 1.071, lr: 0.049975037468784345
epoch: 2000, acc: 0.417, loss: 1.071, lr: 0.04995007490013731
epoch: 3000, acc: 0.417, loss: 1.071, lr: 0.049925137256683606
epoch: 4000, acc: 0.417, loss: 1.071, lr: 0.049900224501110035
epoch: 5000, acc: 0.417, loss: 1.071, lr: 0.04987533659617785
epoch: 6000, acc: 0.417, loss: 1.071, lr: 0.04985047350472258
epoch: 7000, acc: 0.417, loss: 1.071, lr: 0.04982563518965381
epoch: 8000, acc: 0.417, loss: 1.071, lr: 0.04980082161395499
epoch: 9000, acc: 0.417, loss: 1.071, lr: 0.04977603274068329
epoch: 10000, acc: 0.417, loss: 1.071, lr: 0.04975126853296942
```

Figure 13: Optimización AdaGrad con función de activación ReLU

Además, se plantea el siguiente gráfico que representa la pérdida usando el dataset del espiral con la función ReLU

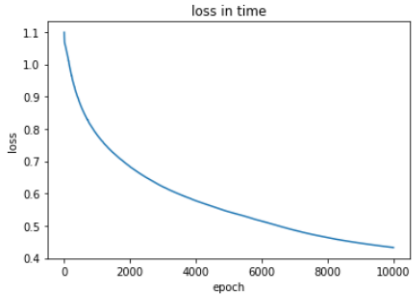


Figure 14: Pérdida del espiral con función ReLU

Finalmente, se probó la red neuronal haciendo uso de este optimizador y la función de activación ReLU. Generando los siguientes resultados:

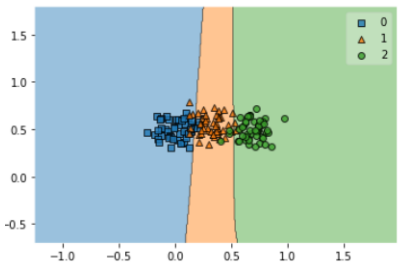


Figure 15: Fronteras de decisión para el dataset de separación lineal

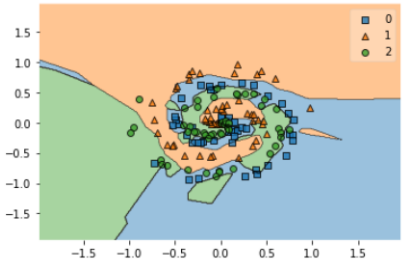


Figure 16: Fronteras de decisión para el dataset del espiral

5.2.3 SDG

- Utilizando el dataset de separación lineal:
  - Lineal

```
epoch: 0, acc: 0.333, loss: 1.098, lr: 0.05
epoch: 1000, acc: 0.930, loss: 0.300, lr: 0.049975037468784345
epoch: 2000, acc: 0.937, loss: 0.186, lr: 0.04995007490013731
epoch: 3000, acc: 0.937, loss: 0.164, lr: 0.049925137256683606
epoch: 4000, acc: 0.940, loss: 0.156, lr: 0.049900224501110035
epoch: 5000, acc: 0.940, loss: 0.153, lr: 0.04987533659617785
epoch: 6000, acc: 0.940, loss: 0.151, lr: 0.04985047350472258
epoch: 7000, acc: 0.940, loss: 0.150, lr: 0.04982563518965381
epoch: 8000, acc: 0.940, loss: 0.150, lr: 0.04980082161395499
epoch: 9000, acc: 0.940, loss: 0.149, lr: 0.04977603274068329
epoch: 10000, acc: 0.940, loss: 0.149, lr: 0.04975126853296942
```

Figure 17: Optimización SGD con función de activación lineal

  - Step

```
epoch: 0, acc: 0.333, loss: 1.108, lr: 0.05
epoch: 1000, acc: 0.927, loss: 0.191, lr: 0.049975037468784345
epoch: 2000, acc: 0.933, loss: 0.178, lr: 0.04995007490013731
epoch: 3000, acc: 0.933, loss: 0.174, lr: 0.049925137256683606
epoch: 4000, acc: 0.933, loss: 0.171, lr: 0.049900224501110035
epoch: 5000, acc: 0.933, loss: 0.169, lr: 0.04987533659617785
epoch: 6000, acc: 0.933, loss: 0.168, lr: 0.04985047350472258
epoch: 7000, acc: 0.933, loss: 0.166, lr: 0.04982563518965381
epoch: 8000, acc: 0.933, loss: 0.165, lr: 0.04980082161395499
epoch: 9000, acc: 0.933, loss: 0.164, lr: 0.04977603274068329
epoch: 10000, acc: 0.933, loss: 0.164, lr: 0.04975126853296942
```

Figure 18: Optimización SGD con función de activación step

  - ReLU

```
epoch: 0, acc: 0.333, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.953, loss: 0.317, lr: 0.049975037468784345
epoch: 2000, acc: 0.960, loss: 0.159, lr: 0.04995007490013731
epoch: 3000, acc: 0.960, loss: 0.132, lr: 0.049925137256683606
epoch: 4000, acc: 0.960, loss: 0.123, lr: 0.049900224501110035
epoch: 5000, acc: 0.957, loss: 0.118, lr: 0.04987533659617785
epoch: 6000, acc: 0.953, loss: 0.116, lr: 0.04985047350472258
epoch: 7000, acc: 0.953, loss: 0.115, lr: 0.04982563518965381
epoch: 8000, acc: 0.953, loss: 0.114, lr: 0.04980082161395499
epoch: 9000, acc: 0.953, loss: 0.113, lr: 0.04977603274068329
epoch: 10000, acc: 0.953, loss: 0.113, lr: 0.04975126853296942
```

Figure 19: Optimización SGD con función de activación ReLU

- Utilizando el dataset del espiral:
  - Lineal

```
epoch: 0, acc: 0.333, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.410, loss: 1.084, lr: 0.049975037468784345
epoch: 2000, acc: 0.420, loss: 1.072, lr: 0.04995007490013731
epoch: 3000, acc: 0.427, loss: 1.071, lr: 0.049925137256683606
epoch: 4000, acc: 0.420, loss: 1.071, lr: 0.049900224501110035
epoch: 5000, acc: 0.417, loss: 1.071, lr: 0.04987533659617785
epoch: 6000, acc: 0.417, loss: 1.071, lr: 0.04985047350472258
epoch: 7000, acc: 0.417, loss: 1.071, lr: 0.04982563518965381
epoch: 8000, acc: 0.417, loss: 1.071, lr: 0.04980082161395499
epoch: 9000, acc: 0.417, loss: 1.071, lr: 0.04977603274068329
epoch: 10000, acc: 0.417, loss: 1.071, lr: 0.04975126853296942
```

Figure 20: Optimización SGD con función de activación lineal

- Step

```
epoch: 0, acc: 0.333, loss: 1.100, lr: 0.05
epoch: 1000, acc: 0.613, loss: 0.857, lr: 0.049975037468784345
epoch: 2000, acc: 0.657, loss: 0.784, lr: 0.04995007490013731
epoch: 3000, acc: 0.677, loss: 0.746, lr: 0.049925137256683606
epoch: 4000, acc: 0.680, loss: 0.721, lr: 0.049900224501110035
epoch: 5000, acc: 0.687, loss: 0.703, lr: 0.04987533659617785
epoch: 6000, acc: 0.693, loss: 0.690, lr: 0.04985047350472258
epoch: 7000, acc: 0.693, loss: 0.679, lr: 0.04982563518965381
epoch: 8000, acc: 0.697, loss: 0.670, lr: 0.04980082161395499
epoch: 9000, acc: 0.693, loss: 0.663, lr: 0.04977603274068329
epoch: 10000, acc: 0.697, loss: 0.656, lr: 0.04975126853296942
```

Figure 21: Optimización SGD con función de activación step

- ReLU

```
epoch: 0, acc: 0.333, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.410, loss: 1.084, lr: 0.049975037468784345
epoch: 2000, acc: 0.420, loss: 1.072, lr: 0.04995007490013731
epoch: 3000, acc: 0.427, loss: 1.071, lr: 0.049925137256683606
epoch: 4000, acc: 0.420, loss: 1.071, lr: 0.049900224501110035
epoch: 5000, acc: 0.417, loss: 1.071, lr: 0.04987533659617785
epoch: 6000, acc: 0.417, loss: 1.071, lr: 0.04985047350472258
epoch: 7000, acc: 0.417, loss: 1.071, lr: 0.04982563518965381
epoch: 8000, acc: 0.417, loss: 1.071, lr: 0.04980082161395499
epoch: 9000, acc: 0.417, loss: 1.071, lr: 0.04977603274068329
epoch: 10000, acc: 0.417, loss: 1.071, lr: 0.04975126853296942
```

Figure 22: Optimización SGD con función de activación ReLU

Además, se plantea el siguiente gráfico que representa la pérdida usando el dataset del espiral con la función ReLU

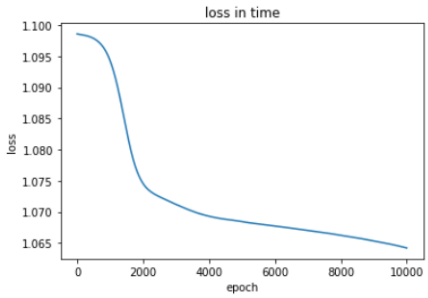


Figure 23: Pérdida del espiral con función ReLU

Finalmente, se probó la red neuronal haciendo uso de este optimizador y la función de activación ReLU. Generando los siguientes resultados:

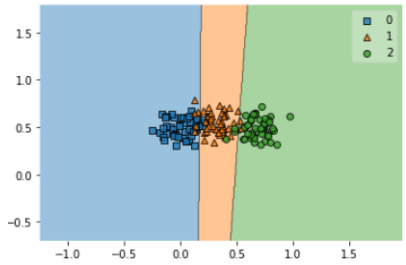


Figure 24: Fronteras de decisión para el dataset de separación lineal

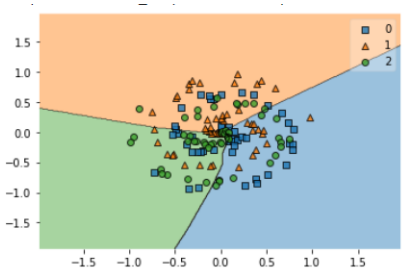


Figure 25: Fronteras de decisión para el dataset del espiral

5.2.4 RMS

- Utilizando el dataset de separación lineal:
  - Lineal

```
epoch: 0, acc: 0.333, loss: 1.098, lr: 0.05
epoch: 1000, acc: 0.927, loss: 0.167, lr: 0.049975037468784345
epoch: 2000, acc: 0.943, loss: 0.149, lr: 0.04995007490013731
epoch: 3000, acc: 0.947, loss: 0.158, lr: 0.049925137256683606
epoch: 4000, acc: 0.943, loss: 0.165, lr: 0.049900224501110035
epoch: 5000, acc: 0.943, loss: 0.149, lr: 0.04987533659617785
epoch: 6000, acc: 0.943, loss: 0.149, lr: 0.04985047350472258
epoch: 7000, acc: 0.943, loss: 0.151, lr: 0.04982563518965381
epoch: 8000, acc: 0.943, loss: 0.149, lr: 0.04980082161395499
epoch: 9000, acc: 0.943, loss: 0.153, lr: 0.04977603274068329
epoch: 10000, acc: 0.940, loss: 0.152, lr: 0.04975126853296942
```

Figure 26: Optimización RMS con función de activación lineal

- Step

```
epoch: 0, acc: 0.320, loss: 1.103, lr: 0.05
epoch: 1000, acc: 0.963, loss: 0.179, lr: 0.049975037468784345
epoch: 2000, acc: 0.967, loss: 0.163, lr: 0.04995007490013731
epoch: 3000, acc: 0.967, loss: 0.152, lr: 0.049925137256683606
epoch: 4000, acc: 0.967, loss: 0.150, lr: 0.049900224501110035
epoch: 5000, acc: 0.967, loss: 0.144, lr: 0.04987533659617785
epoch: 6000, acc: 0.967, loss: 0.144, lr: 0.04985047350472258
epoch: 7000, acc: 0.967, loss: 0.143, lr: 0.04982563518965381
epoch: 8000, acc: 0.967, loss: 0.143, lr: 0.04980082161395499
epoch: 9000, acc: 0.967, loss: 0.143, lr: 0.04977603274068329
epoch: 10000, acc: 0.967, loss: 0.142, lr: 0.04975126853296942
```

Figure 27: Optimización RMS con función de activación step

- ReLU

```
epoch: 0, acc: 0.333, loss: 1.098, lr: 0.05
epoch: 1000, acc: 0.950, loss: 0.105, lr: 0.049975037468784345
epoch: 2000, acc: 0.960, loss: 0.094, lr: 0.04995007490013731
epoch: 3000, acc: 0.960, loss: 0.094, lr: 0.049925137256683606
epoch: 4000, acc: 0.960, loss: 0.083, lr: 0.049900224501110035
epoch: 5000, acc: 0.960, loss: 0.122, lr: 0.04987533659617785
epoch: 6000, acc: 0.957, loss: 0.082, lr: 0.04985047350472258
epoch: 7000, acc: 0.957, loss: 0.082, lr: 0.04982563518965381
epoch: 8000, acc: 0.957, loss: 0.081, lr: 0.04980082161395499
epoch: 9000, acc: 0.967, loss: 0.077, lr: 0.04977603274068329
epoch: 10000, acc: 0.957, loss: 0.086, lr: 0.04975126853296942
```

Figure 28: Optimización RMS con función de activación ReLU

- Utilizando el dataset del espiral:
  - Lineal

```
epoch: 0, acc: 0.320, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.427, loss: 1.071, lr: 0.049975037468784345
epoch: 2000, acc: 0.417, loss: 1.071, lr: 0.04995007490013731
epoch: 3000, acc: 0.417, loss: 1.072, lr: 0.049925137256683606
epoch: 4000, acc: 0.423, loss: 1.075, lr: 0.049900224501110035
epoch: 5000, acc: 0.433, loss: 1.071, lr: 0.04987533659617785
epoch: 6000, acc: 0.440, loss: 1.071, lr: 0.04985047350472258
epoch: 7000, acc: 0.383, loss: 1.091, lr: 0.04982563518965381
epoch: 8000, acc: 0.390, loss: 1.071, lr: 0.04980082161395499
epoch: 9000, acc: 0.393, loss: 1.125, lr: 0.04977603274068329
epoch: 10000, acc: 0.420, loss: 1.071, lr: 0.04975126853296942
```

Figure 29: Optimización RMS con función de activación lineal

  - Step

```
epoch: 0, acc: 0.333, loss: 1.098, lr: 0.05
epoch: 1000, acc: 0.700, loss: 0.693, lr: 0.049975037468784345
epoch: 2000, acc: 0.717, loss: 0.677, lr: 0.04995007490013731
epoch: 3000, acc: 0.717, loss: 0.663, lr: 0.049925137256683606
epoch: 4000, acc: 0.723, loss: 0.641, lr: 0.049900224501110035
epoch: 5000, acc: 0.743, loss: 0.586, lr: 0.04987533659617785
epoch: 6000, acc: 0.730, loss: 0.634, lr: 0.04985047350472258
epoch: 7000, acc: 0.743, loss: 0.631, lr: 0.04982563518965381
epoch: 8000, acc: 0.717, loss: 0.621, lr: 0.04980082161395499
epoch: 9000, acc: 0.740, loss: 0.554, lr: 0.04977603274068329
epoch: 10000, acc: 0.743, loss: 0.546, lr: 0.04975126853296942
```

Figure 30: Optimización RMS con función de activación step

  - ReLU

```
epoch: 0, acc: 0.320, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.427, loss: 1.071, lr: 0.049975037468784345
epoch: 2000, acc: 0.417, loss: 1.071, lr: 0.04995007490013731
epoch: 3000, acc: 0.417, loss: 1.072, lr: 0.049925137256683606
epoch: 4000, acc: 0.423, loss: 1.075, lr: 0.049900224501110035
epoch: 5000, acc: 0.433, loss: 1.071, lr: 0.04987533659617785
epoch: 6000, acc: 0.440, loss: 1.071, lr: 0.04985047350472258
epoch: 7000, acc: 0.383, loss: 1.091, lr: 0.04982563518965381
epoch: 8000, acc: 0.390, loss: 1.071, lr: 0.04980082161395499
epoch: 9000, acc: 0.393, loss: 1.125, lr: 0.04977603274068329
epoch: 10000, acc: 0.420, loss: 1.071, lr: 0.04975126853296942
```

Figure 31: Optimización RMS con función de activación ReLU

Además, se plantea el siguiente gráfico que representa la pérdida usando el dataset del espiral con la función ReLU

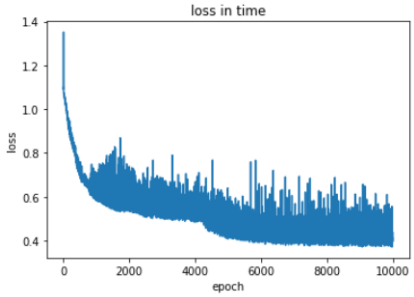


Figure 32: Pérdida del espiral con función ReLU

Finalmente, se probó la red neuronal haciendo uso de este optimizador y la función de activación ReLU. Generando los siguientes resultados:

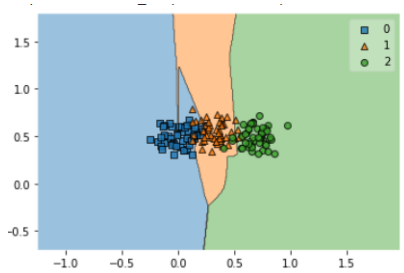


Figure 33: Fronteras de decisión para el dataset de separación lineal

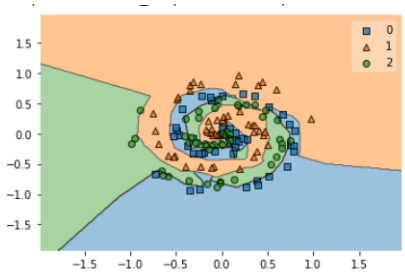


Figure 34: Fronteras de decisión para el dataset del espiral

5.2.5 Adam

- Utilizando el dataset de separación lineal:
  - Lineal

```
epoch: 0, acc: 0.497, loss: 1.098, lr: 0.05
epoch: 1000, acc: 0.943, loss: 0.149, lr: 0.049975037468784345
epoch: 2000, acc: 0.943, loss: 0.149, lr: 0.04995007490013731
epoch: 3000, acc: 0.940, loss: 0.149, lr: 0.049925137256683606
epoch: 4000, acc: 0.943, loss: 0.149, lr: 0.049900224501110035
epoch: 5000, acc: 0.943, loss: 0.149, lr: 0.04987533659617785
epoch: 6000, acc: 0.943, loss: 0.149, lr: 0.04985047350472258
epoch: 7000, acc: 0.940, loss: 0.151, lr: 0.04982563518965381
epoch: 8000, acc: 0.943, loss: 0.149, lr: 0.04980082161395499
epoch: 9000, acc: 0.943, loss: 0.149, lr: 0.04977603274068329
epoch: 10000, acc: 0.943, loss: 0.149, lr: 0.04975126853296942
```

Figure 35: Optimización Adam con función de activación lineal

- Step

```
epoch: 0, acc: 0.337, loss: 1.093, lr: 0.05
epoch: 1000, acc: 0.953, loss: 0.108, lr: 0.049975037468784345
epoch: 2000, acc: 0.953, loss: 0.106, lr: 0.04995007490013731
epoch: 3000, acc: 0.953, loss: 0.105, lr: 0.049925137256683606
epoch: 4000, acc: 0.953, loss: 0.112, lr: 0.049900224501110035
epoch: 5000, acc: 0.953, loss: 0.105, lr: 0.04987533659617785
epoch: 6000, acc: 0.953, loss: 0.105, lr: 0.04985047350472258
epoch: 7000, acc: 0.953, loss: 0.105, lr: 0.04982563518965381
epoch: 8000, acc: 0.953, loss: 0.105, lr: 0.04980082161395499
epoch: 9000, acc: 0.953, loss: 0.105, lr: 0.04977603274068329
epoch: 10000, acc: 0.953, loss: 0.105, lr: 0.04975126853296942
```

Figure 36: Optimización Adam con función de activación step

- ReLU

```
epoch: 0, acc: 0.333, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.953, loss: 0.112, lr: 0.049975037468784345
epoch: 2000, acc: 0.950, loss: 0.112, lr: 0.04995007490013731
epoch: 3000, acc: 0.953, loss: 0.112, lr: 0.049925137256683606
epoch: 4000, acc: 0.953, loss: 0.112, lr: 0.049900224501110035
epoch: 5000, acc: 0.953, loss: 0.112, lr: 0.04987533659617785
epoch: 6000, acc: 0.950, loss: 0.113, lr: 0.04985047350472258
epoch: 7000, acc: 0.953, loss: 0.112, lr: 0.04982563518965381
epoch: 8000, acc: 0.950, loss: 0.112, lr: 0.04980082161395499
epoch: 9000, acc: 0.953, loss: 0.112, lr: 0.04977603274068329
epoch: 10000, acc: 0.953, loss: 0.112, lr: 0.04975126853296942
```

Figure 37: Optimización Adam con función de activación ReLU

- Utilizando el dataset del espiral:
  - Lineal

```
epoch: 0, acc: 0.333, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.417, loss: 1.071, lr: 0.049975037468784345
epoch: 2000, acc: 0.417, loss: 1.071, lr: 0.04995007490013731
epoch: 3000, acc: 0.417, loss: 1.071, lr: 0.049925137256683606
epoch: 4000, acc: 0.417, loss: 1.071, lr: 0.049900224501110035
epoch: 5000, acc: 0.417, loss: 1.071, lr: 0.04987533659617785
epoch: 6000, acc: 0.417, loss: 1.071, lr: 0.04985047350472258
epoch: 7000, acc: 0.417, loss: 1.071, lr: 0.04982563518965381
epoch: 8000, acc: 0.417, loss: 1.071, lr: 0.04980082161395499
epoch: 9000, acc: 0.417, loss: 1.071, lr: 0.04977603274068329
epoch: 10000, acc: 0.417, loss: 1.071, lr: 0.04975126853296942
```

Figure 38: Optimización Adam con función de activación lineal

- Step

```
epoch: 0, acc: 0.330, loss: 1.102, lr: 0.05
epoch: 1000, acc: 0.783, loss: 0.484, lr: 0.049975037468784345
epoch: 2000, acc: 0.783, loss: 0.477, lr: 0.04995007490013731
epoch: 3000, acc: 0.780, loss: 0.475, lr: 0.049925137256683606
epoch: 4000, acc: 0.783, loss: 0.474, lr: 0.049900224501110035
epoch: 5000, acc: 0.783, loss: 0.474, lr: 0.04987533659617785
epoch: 6000, acc: 0.783, loss: 0.474, lr: 0.04985047350472258
epoch: 7000, acc: 0.783, loss: 0.474, lr: 0.04982563518965381
epoch: 8000, acc: 0.783, loss: 0.473, lr: 0.04980082161395499
epoch: 9000, acc: 0.783, loss: 0.473, lr: 0.04977603274068329
epoch: 10000, acc: 0.787, loss: 0.473, lr: 0.04975126853296942
```

Figure 39: Optimización Adam con función de activación step

- ReLU

```
epoch: 0, acc: 0.333, loss: 1.099, lr: 0.05
epoch: 1000, acc: 0.417, loss: 1.071, lr: 0.049975037468784345
epoch: 2000, acc: 0.417, loss: 1.071, lr: 0.04995007490013731
epoch: 3000, acc: 0.417, loss: 1.071, lr: 0.049925137256683606
epoch: 4000, acc: 0.417, loss: 1.071, lr: 0.049900224501110035
epoch: 5000, acc: 0.417, loss: 1.071, lr: 0.04987533659617785
epoch: 6000, acc: 0.417, loss: 1.071, lr: 0.04985047350472258
epoch: 7000, acc: 0.417, loss: 1.071, lr: 0.04982563518965381
epoch: 8000, acc: 0.417, loss: 1.071, lr: 0.04980082161395499
epoch: 9000, acc: 0.417, loss: 1.071, lr: 0.04977603274068329
epoch: 10000, acc: 0.417, loss: 1.071, lr: 0.04975126853296942
```

Figure 40: Optimización Adam con función de activación ReLU

Además, se plantea el siguiente gráfico que representa la pérdida usando el dataset del espiral con la función ReLU

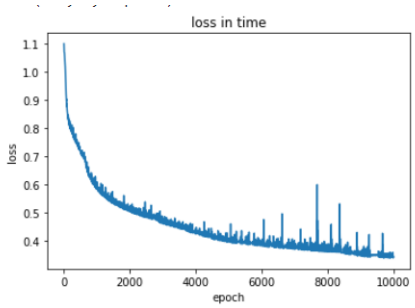


Figure 41: Pérdida del espiral con función ReLU

Finalmente, se probó la red neuronal haciendo uso de este optimizador y la función de activación ReLU. Generando los siguientes resultados:

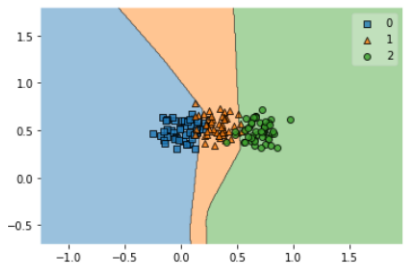


Figure 42: Fronteras de decisión para el dataset de separación lineal

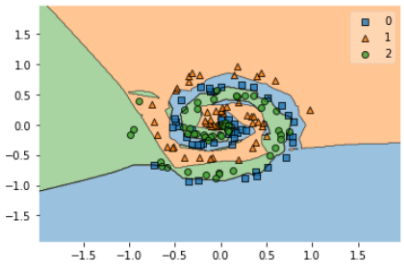


Figure 43: Fronteras de decisión para el dataset del espiral

6. Conclusiones

- Para el dataset de separación lineal se evidenció que no existe una diferencia significativa en los resultados a medida que se cambian los optimizadores y las funciones de optimización.
- Por su parte, para el dataset del espiral se indica que el resultado mas cercano a la realidad se dió al usar el optimizador Adam junto con la función de activación ReLU.
- Para trabajos futuros, se recomienda usar batches al realizar el proceso de aprendizaje de la red neuronal. Esto debido a que probablemente se presentó overfitting en este proceso presentando así una solución que quizá no se asemeja tanto a la realidad como se quisiera.
- Se recomienda a su vez trabajar con la función de activación sigmoide o alteraciones de la ReLU como lo son la Leaky ReLU y la REU. Considerando su similitud con la ReLU y tomando en cuenta que esta última fue la función dominante para este proyecto.
- Si bien el código funciona para datos n-dimensionales, hay algunas cosas que no realiza de manera correcta como son problemas que incluyen regresión binaria.
- La función de pérdida utilizada en este proyecto está enfocada a problemas de datos bivariantes con una clase. Por lo tanto, se recomienda usar otra función de pérdida en caso de trabajar con datos de diferente tipo.

References

[1] Sanket Doshi. Various optimization algorithms for training neural network, 2019.

[2] Kevin Gurney. *An introduction to neural networks*. UCL Press Limited, 1997.

[3] Daniel Kukiela Harrison Kinsley. *Neural Networks from Scratch in Python*. Kinsley Enterprises, 2020.

[4] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.