Actividad: Análisis y Reporte sobre el desempeño del modelo.

Inteligencia Artificial para la ciencia de datos

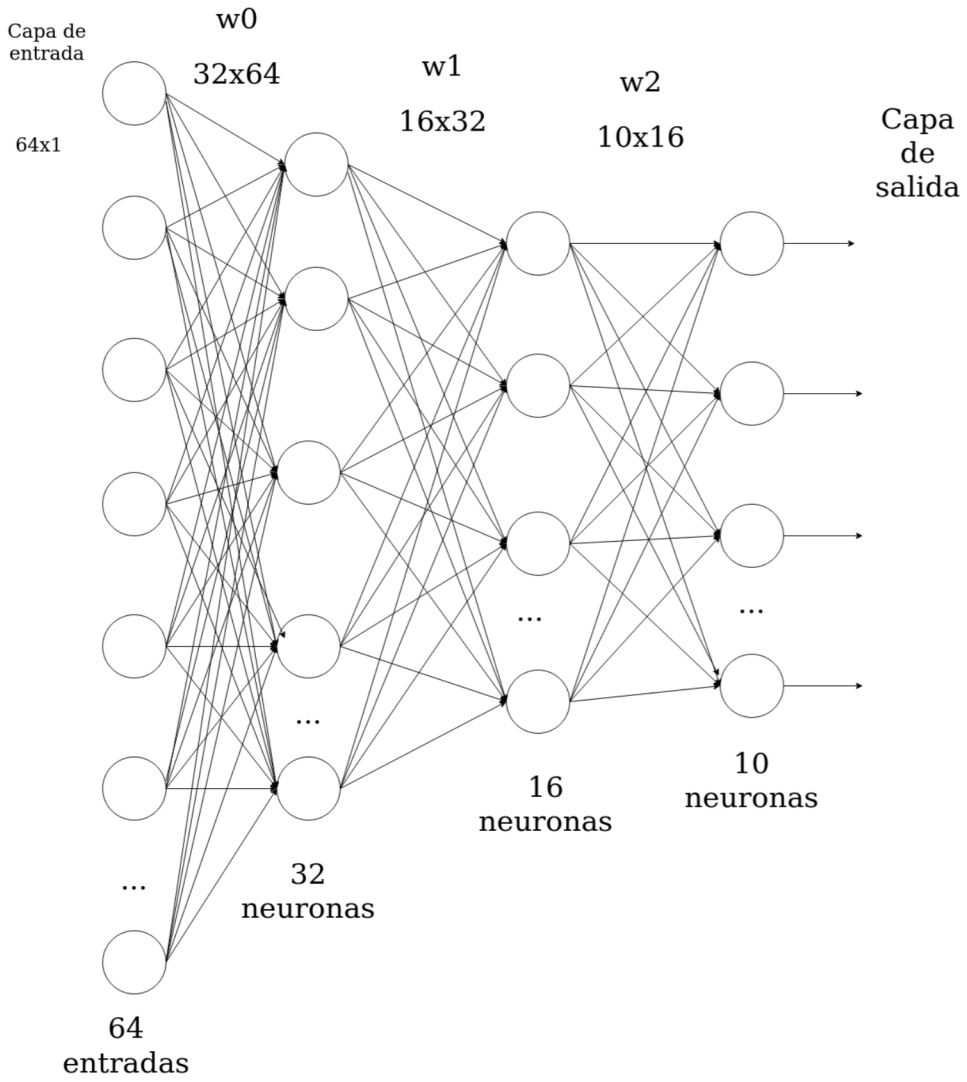
Leonardo Gracida Muñoz A01379812

Profesor: Jorge Adolfo Ramírez Uresti

Fecha de entrega: Martes 13 de septiembre de 2022

El dataset utilizado fue uno ya precargado en scikit learn, son imágenes de 8x8, son imágenes de dígitos o números escritos a mano, que van del 0 al 9. El target de este dataset es el número que representa o el label de cada imagen que son 0,1,2,3,4,5,6,7,8,9.

Para poder predecir el target o label de cada imagen lo que vamos a utilizar es una red neuronal con la siguiente estructura:



Lo que primero debemos hacer es normalizar los datos de la imágenes o números que conforman la matriz, dividimos cada una de las matrices o imágenes del dataset entre el número máximo que tienen todas las matrices, en este caso es 16, esto lo hacemos para normalizar la entrada para que las matrices vayan de 0 a 1. Al ya tener estas matrices normalizadas las aplanamos de una forma de 8X8 a una forma de 64x1, luego usamos dos hidden layers, una de 32 neuronas y otra de 16 neuronas, finalizando con una capa de salida de 10 neuronas.

En cuestión de los pesos de la red vamos a utilizar numpy para poder ingresar todos los pesos o weights en una matriz y usar la multiplicación matricial y la sigmoide para pasar por cada una de las capas hasta la capa final de la red. Como también vamos a usar funciones de numpy para poder actualizar todos los pesos durante el proceso de entrenamiento, además de separar la data en una parte de train, test y validation.

Las salidas de la red las vamos a decodificar de la siguiente manera, como son diez salidas, y son diez labels, vamos a declarar una neurona a cada una de ellas, por ejemplo, 0 = 10000000000, 1 = 0100000000, …, 9 = 0000000001.

Para poder entrenar la red vamos a utilizar el algoritmo de Back Propagation usando las siguientes fórmulas:

- Error de la capa de salida:

$$\delta_k = O_k(1 - O_k)(t_k - O_k)$$

- Error de las capas ocultas:

$$\delta_h = O_h(1 - O_h) \sum_{i=j}^{n} w_{jh}\delta_k$$

- Delta w o actualización del peso:

$$\Delta w = n * \delta_z * x_i$$

- Actualización del peso:

$$w_i = w + \alpha * \Delta w$$

Las neuronas que vamos a usar para esta red es un perceptrón con una función activación de tipo sigmoide. La cual suma todas las entradas multiplicadas por sus respectivos pesos para ingresar la entrada en la función de activación y obtener una salida que va de cero a uno.

En nuestro análisis vamos a separar la dataset en un 30% de test y 70% de train inicialmente y luego vamos a dividir el train otra vez extrayendo 10% de validation. Obteniendo un total de muestras en cada una de las divisiones.

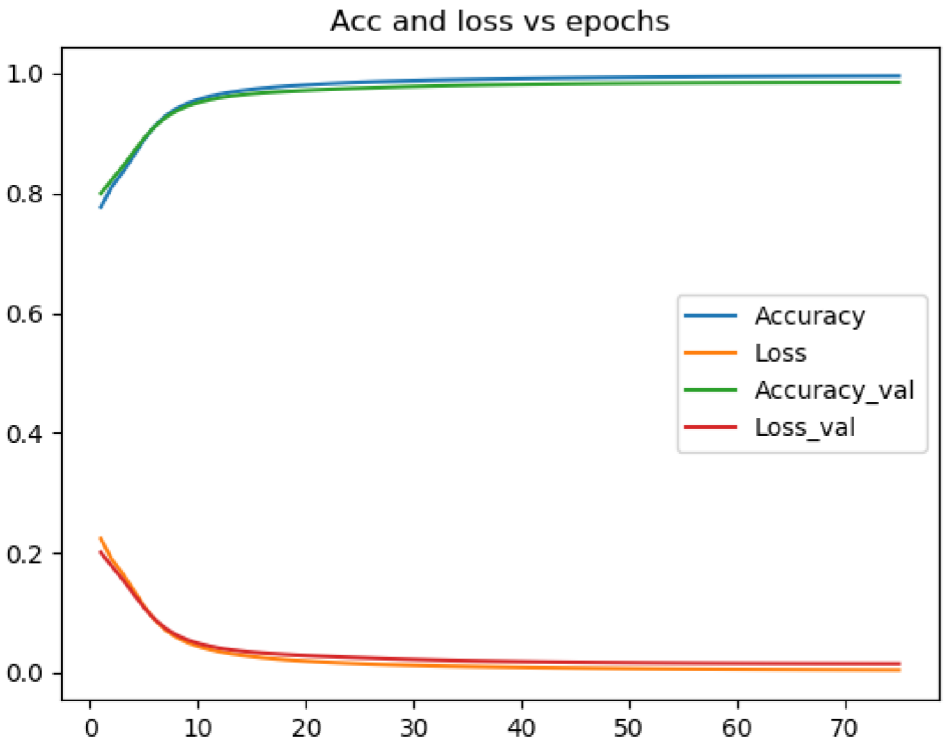
Muestras train: 1295

Muestras test: 359

Muestras validation: 143

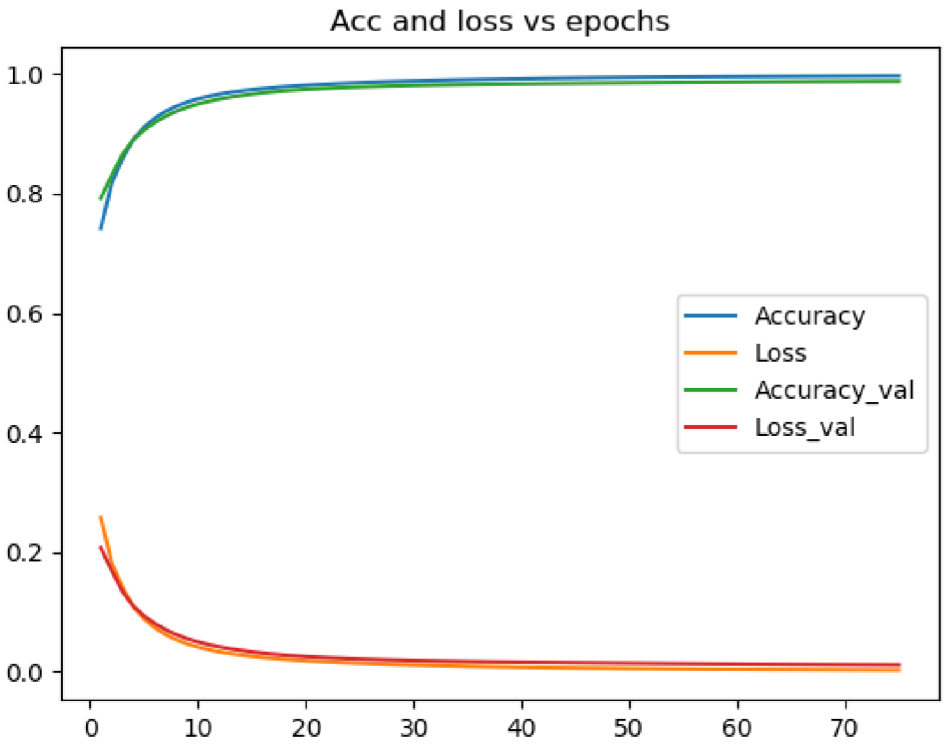
Hicimos diferentes pruebas para ver que separación es la más óptima, lo separamos en 30%. 40% y 20% de test. En estas pruebas usamos el mismo validation de 10% de lo que se extrae de train al separarlos una vez, como 75 epochs y un learning rate de 0.1.

- 30% de test:



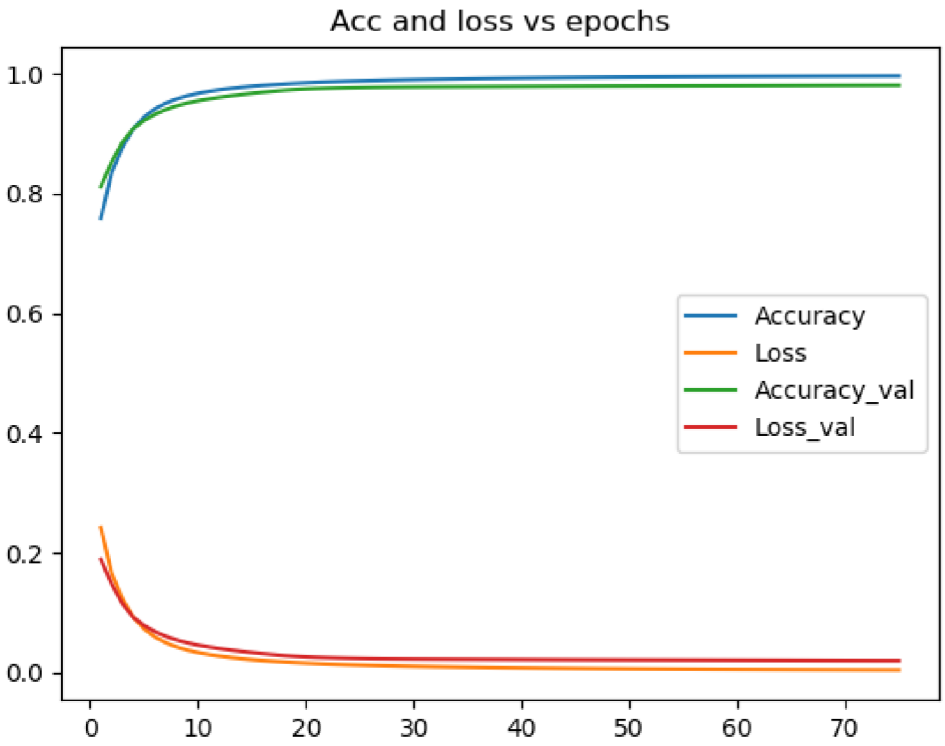
Prueba test: acc: 97.08 ======== loss: 0.029

- 40% de test:



Prueba test: acc: 96.52 ======== loss: 0.034

- 20% de test:



Prueba test: acc: 96.25 ' ======== loss: 0.0374

Al ver el resultado con las diferentes proporciones, el comportamiento del entrenamiento de la red es muy bueno o muy estable, no hay picos, y se acerca al resultado esperado de manera aceptable, teniendo una buena acc y loss, al final vemos que la mejor proporción de test es la de 30%, ya que al hacer 20% entramos un poco en overfitting ya que tiene un acc menor al tener 30% a pesar de ser menos muestras, y usando 40% entramos en underfitting ya que también tenemos menor acc, todas dan buenos resultados al entrenar y probar, no sólo en acc sino en las gráficas, es por eso que el usado finalmente es 30% de test.

Al ver el resultado de nuestro entrenamiento y pruebas, al ver la acc y loss, vemos que nuestro bias y varianza es pequeño, al ver que nuestra pérdida es muy pequeña en las diferentes pruebas hechas anteriormente, como también al ver que el entrenamiento es muy uniforme, al no tener picos aparentes que hagan crecer y bajar tanto la acc como la loss a lo largo de las epochs. Demostrando que las predicciones siempre son cercanas a la ideal en la salida de la red viendo la loss final en el test, y en caso de tener un error no hace una predicción tan diferente o alejada de la predicción correcta. Teniendo el mismo comportamiento y resultados parecidos al hacer el train y validation.

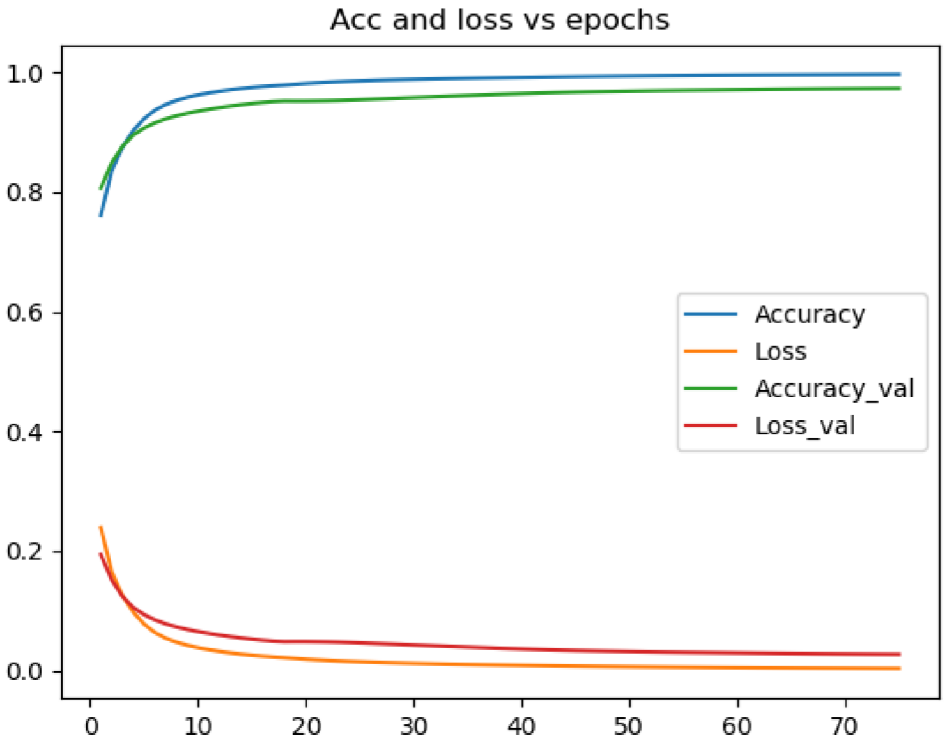
Nuestro modelo no presenta tener nada de ovefitting, ni underfitting, ya que logra tener una buena acc y loss en las pruebas de entrenamiento y testeo, logrando estar arriba de 0.94% de acc y teniendo una loss menor a 0.2 o muy cercana a cero, puede que al inicio al ver los

resultados del train, es normal que se infiere que el modelo está sobre entrenado, ya que el tener una acc alta en la parte del train puede significar que ya se aprendió las entradas de memoria, pero al observar el comportamiento de la parte de validation y al hacer la prueba final en el test vemos que esto no es así, mostrando muy buenos resultados, teniendo casi todas las predicciones bien.

- Prueba de cambio de hiper parámetros para el mejoramiento del modelo:

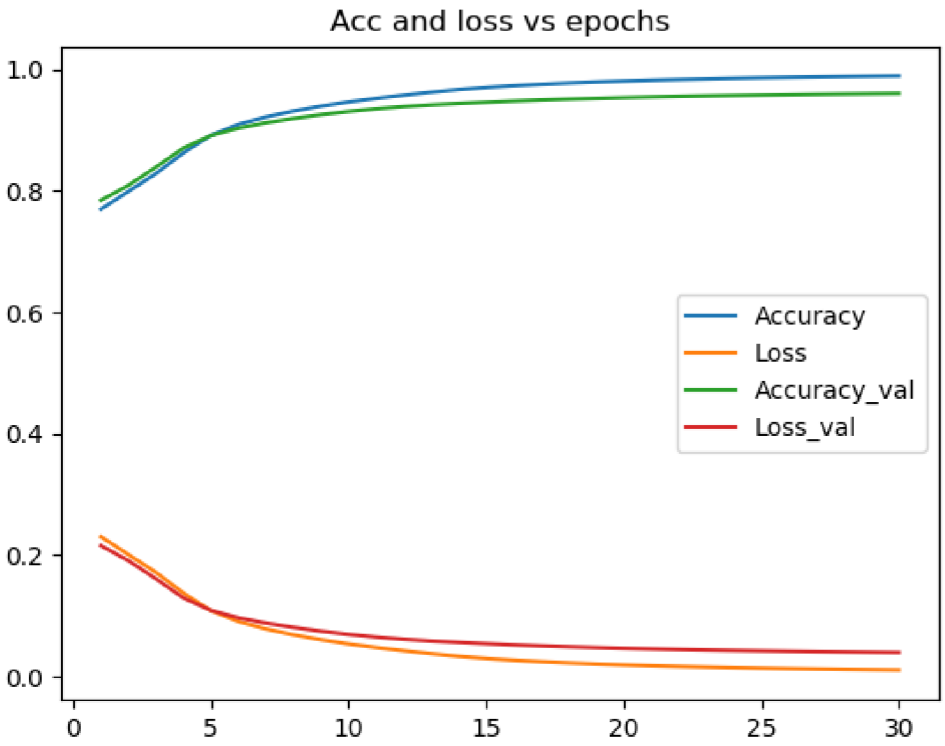
La última prueba que vamos a hacer es el mover los hiperparametros de nuestro modelo para poder ver si nuestro modelo, mejora o empeora, o si es mejor dejarlo con los mismo hiper parámetros. Los hiperparametros a mover es learning rate y las epochs, en estas pruebas vamos a usar un test de 30%.

- Epochs: 75, Lr: 0.1:



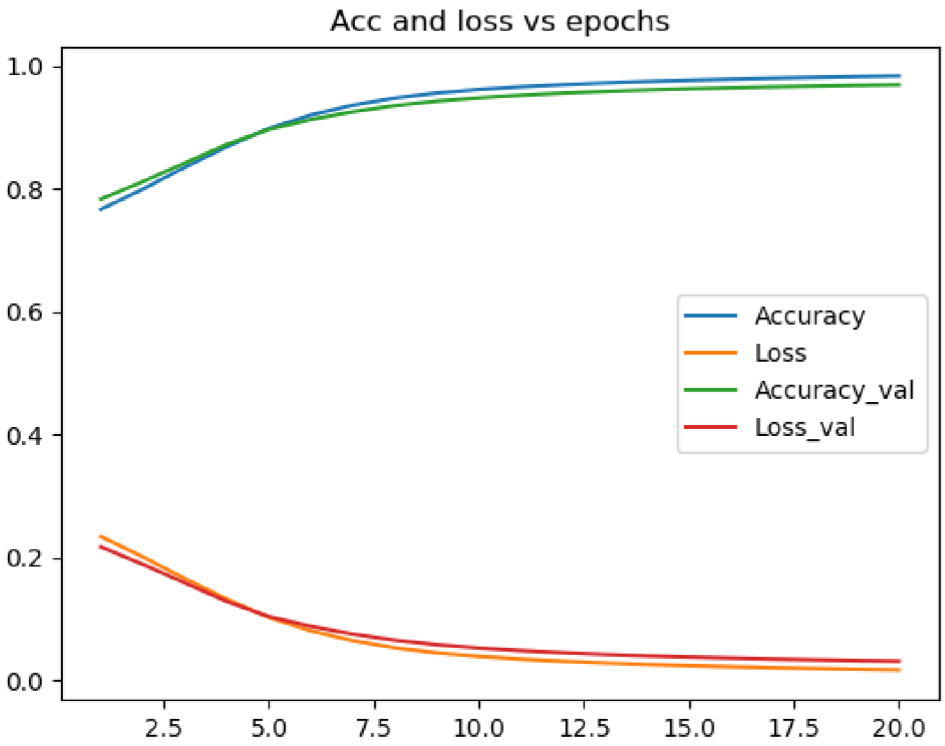
Prueba test: 'acc:', 97.98, ' ======== loss: 0.02

- Epochs: 30, Lr: 0.1:



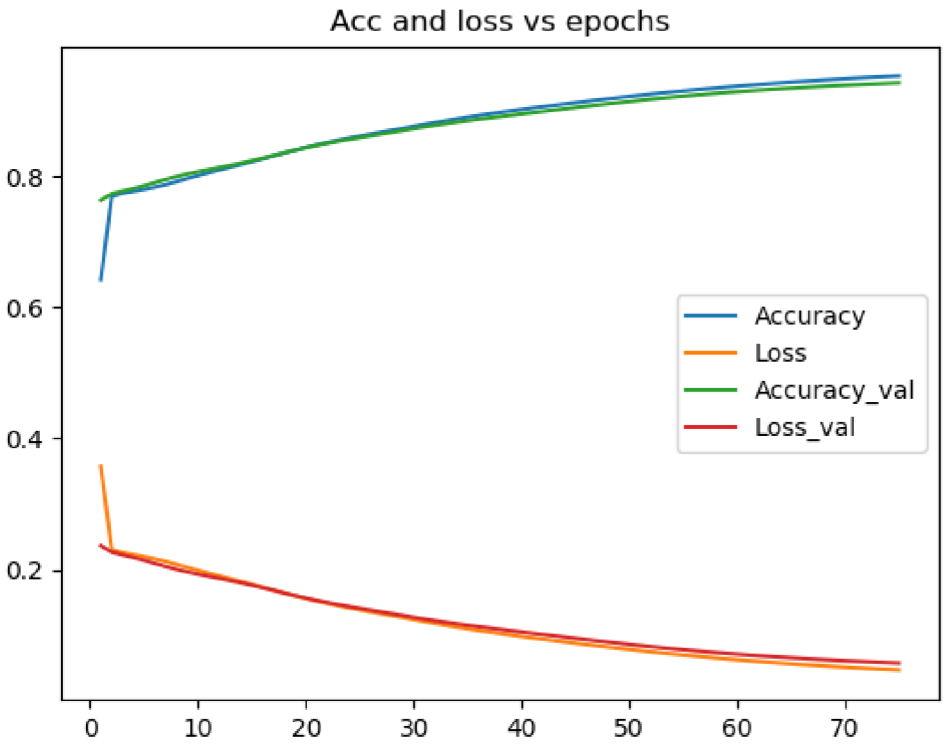
Prueba test: acc: 98.27 ======== loss: 0.017

- Epochs: 20, Lr: 0.1:



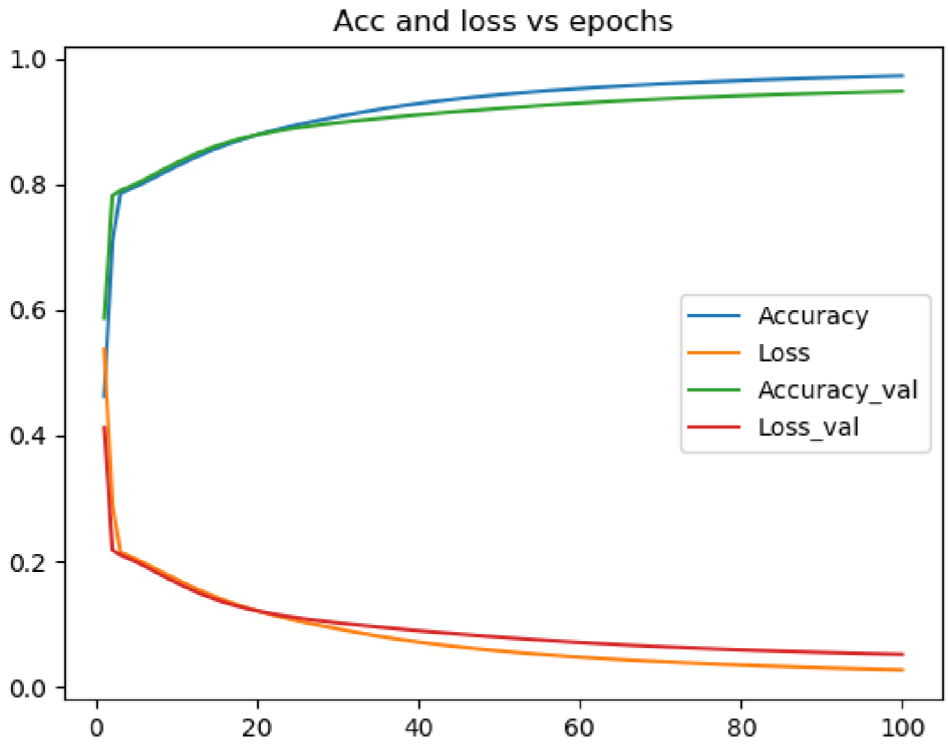
Prueba test: acc: 96.76 ======== loss: 0.03

- Epochs: 75, Lr: 0.01:



Prueba test: acc: 93.02 ======== loss: 0.069

- Epochs: 100, Lr: 0.01:

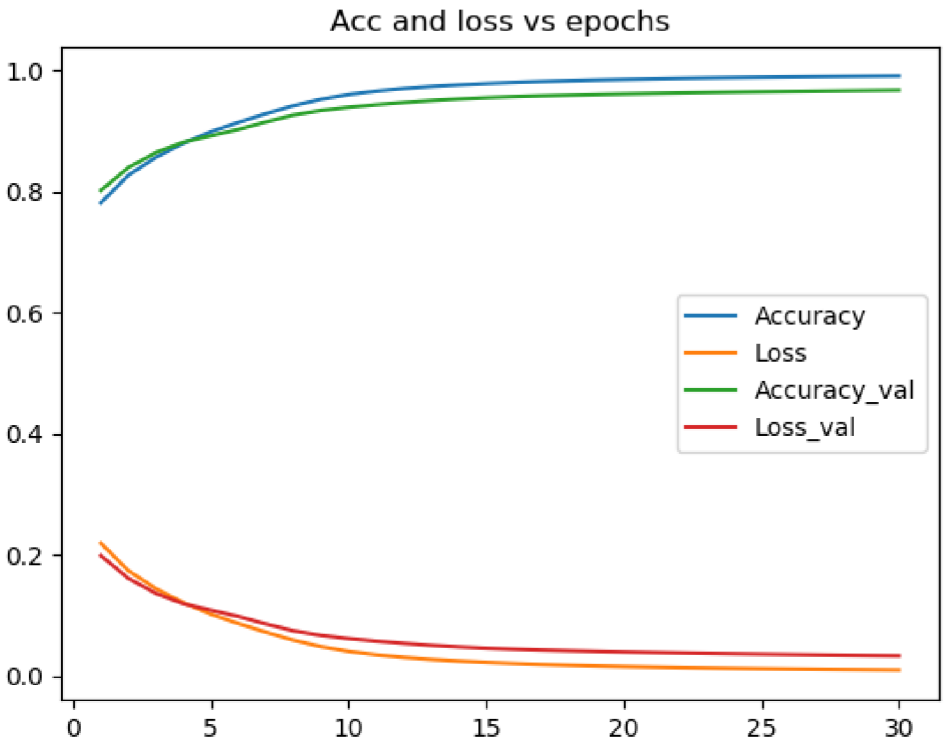


Prueba test: acc: 94.97 ========= loss: 0.05

Como podemos observar nuestro modelo mejoró su precisión al reducir su número de epochs, esto mostrando que aunque no estábamos dentro de un área de overfitting, ya estábamos acercándonos a esta, ya que al hacer menos epochs el modelo pudo aprender mejor y tener una mejor precisión al momento de analizar las muestras del test. En cuestión del learning rate vemos que el usar uno pequeño al entrenar este modelo no es una muy buena idea, ya que necesitamos una cantidad mucho más grande de epochs para poder obtener la misma precisión que al usar uno de 0.1, mostrando que este es mucho más eficiente que el pequeño, demostrando que el encontrar el mínimo global no es muy difícil, y que no necesitamos dar pasos tan pequeños para poder encontrarlo.

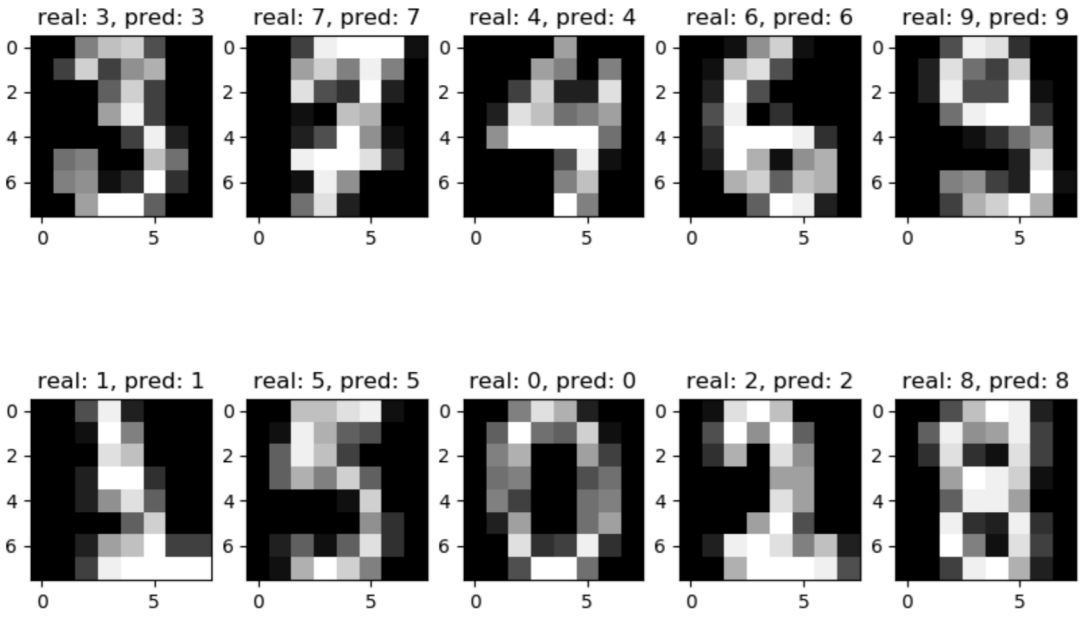
Finalmente al hacer todas estas pruebas determinamos que los parámetros que tienen un mejor desempeño para nuestro modelo es el usar un 30% de test, una cantidad de 30 epochs y un learning rate de 0.1.

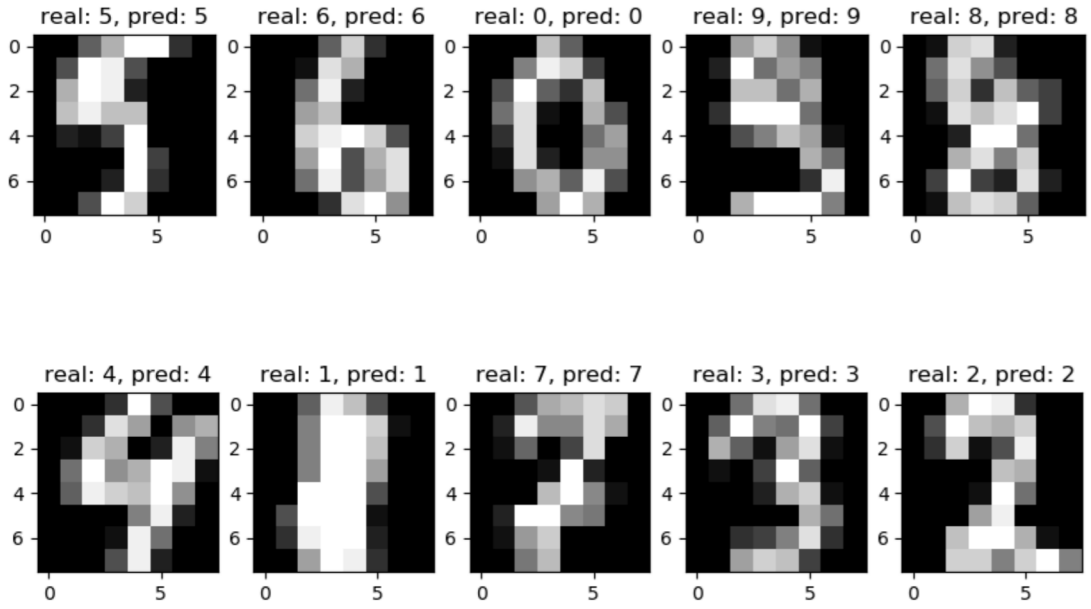
- Resultados prueba final:



Prueba test: acc: 98.1 ======== loss: 0.02

Ahora mostramos una cuantas predicciones del modelo para poder demostrar su efectividad:





El código que entrena el modelo, lo guarda y en el que se puede modificar los hiper parámetros es el llamado train.py, el que prueba el modelo en la parte de test es el llamado test.py, estos códigos generan de manera automática las gráficas usadas en el documento como también la imagen con las predicciones hechas.