Tarea N° 2:

Redes Neuronales

Manuel Muñoz M.

COM4402 – Introducción a Inteligencia Artificial

Escuela de Ingeniería, Universidad de O'Higgins

28, 09, 2023

***Abstract*— Las redes neuronales son un tipo de aprendizaje supervisado que permiten tomar distintas decisiones. En este informe, se utilizará este modelo para la clasificación de Dígitos Manuscritos, probando Redes con distintas capas, neuronas y funciones de activación con el fin de comparar y encontrar la mejor para solucionar este tipo de problemas. Los modelos de redes neuronales que planeamos entrenar se componen de la siguiente manera: tienen una capa inicial que recibe datos de entrada con 64 características o dimensiones. Luego, pueden incluir una o dos capas intermedias ocultas, que procesan y transforman esta información. Finalmente, cuentan con una capa de salida que consta de 10 neuronas, lo que sugiere que están diseñadas para tareas en las que se deben realizar clasificaciones en 10 categorías o clases diferentes. En particular, se utilizará Python y Pytorch para realizar todo este procedimiento. En cuanto a los resultados obtenidos, los distintos modelos obtuvieron muy buenos resultados, lás metricas accuracy, precision, recall y f1_score estuvieron por sobre el valor 0.93, lo cual es algo demasiado bueno. El mejor modelo con el que se entrenó la Red, fue con 2 capas ocultas, 40 y 40 neuronas cada una utilizando como función de activación ReLu, en particular esta red obtuvo valores en las métricas cercanos a 0.96.**

***Keywords*— Redes neuronales, Pytorch, clasificación usando Redes neuronales.**

I. INTRODUCCION

Las Redes Neuronales son un método de aprendizaje supervisado que ha tomado mucha fuerza en la computación, sobre todo últimamente ya que se cuenta con un gran poder de cómputo con respecto a los inicios de la computación.

El objetivo de este informe es utilizar estas redes en un problema de clasificación de dígitos, el conjunto de datos a utilizar es el Optical Recognition of Handwritten Digits Data Set. Este conjunto tiene 64 características, con 10 clases y 5620 muestras en total.

Las redes que vamos a entrenar tienen la siguiente estructura: una capa de entrada con 64 dimensiones para los datos de entrada, una o dos capas ocultas, y una capa de salida con 10 neuronas que utiliza la función de activación softmax. La función de pérdida que utilizaremos es la entropía cruzada, y el optimizador que usaremos es el algoritmo Adam. Es importante destacar que la función softmax está implícita en la función de pérdida CrossEntropyLoss de PyTorch, por lo que no es necesario agregar una capa softmax en la salida de la red.

Se irá probando cada una de estas redes con diferentes modelos, a través de una comparación exhaustiva. Además de utilizar un "early stopping" que significa detener el entrenamiento de la red cuando el loss del conjunto de validación comience a aumentar mientras el de entrenamiento siga bajando, de esta forma se evita el overfitting de la red neuronal que estamos entrenando.

En cuanto a la tecnología a utilizar, se utilizará PyTorch para entrenar y validar la red neuronal que implementa un clasificador de dígitos.

Analizaremos, como afecta cambiar el tamaño de la red, es decir, el número de capas ocultas y la cantidad de neuronas en esas capas, así como la elección de la función de activación, a través de la matriz de confusion y distintas métricas de aprendizaje supervisado para evaluar el mejor modelo, como lo son accuracy, recall, precision y f1_score.

II. MARCO TEORICO

. Las redes neuronales, también denominadas perceptrones multicapa o RNA, representan un enfoque de aprendizaje automático que se inspira en el funcionamiento del cerebro humano. Su estructura se basa en la interconexión de neuronas dispuestas en capas, lo que incluye una capa de entrada, capas ocultas y una capa de salida. A través de un proceso de entrenamiento, estas redes ajustan

sus conexiones y parámetros con el objetivo de minimizar las discrepancias entre las predicciones generadas y los resultados deseados.

Un aspecto destacado de las redes neuronales es su capacidad para identificar patrones complejos en datos y realizar tareas de procesamiento de información. Esta versatilidad las ha convertido en herramientas valiosas en una amplia gama de aplicaciones de aprendizaje automático.

Existen siete tipos diferentes de redes neuronales que pueden ser utilizadas:

Perceptrón Multicapa (MLP): Es una red neuronal feedforward que consta de tres o más capas, incluyendo una capa de entrada, una o más capas ocultas y una capa de salida. Emplea funciones de activación no lineales.

Red Neuronal Convolucional (CNN): Diseñada para procesar datos de entrada con una estructura de cuadrícula, como imágenes. Utiliza capas convolucionales y de agrupación para extraer características de los datos de entrada.

Red Neuronal Recursiva (RNN): Capaz de operar con secuencias de entrada de longitud variable, como texto. Utiliza conexiones ponderadas para hacer predicciones estructuradas.

Red Neuronal Recurrente (RNN): Este tipo de red establece conexiones entre las neuronas en un ciclo dirigido, lo que le permite procesar datos secuenciales.

Memoria a Corto Plazo (LSTM): Una variante de las RNN diseñada para abordar el problema del gradiente que desaparece durante el entrenamiento. Emplea celdas y puertas de memoria para leer, escribir y borrar información selectivamente.

Secuencia a Secuencia (Seq2Seq): Utiliza dos RNN para mapear secuencias de entrada a secuencias de salida, como en el caso de la traducción de un idioma a otro.

Red Neuronal Superficial: Consiste en una red neuronal con una sola capa oculta, a menudo utilizada para tareas más simples o como un componente básico en redes neuronales más grandes.

¿Qué son las funciones de activación?

Las funciones de activación se encargan de que haya no linealidad en el modelo, esto hace que las redes neuronales tengan la capacidad de identificar patrones y relaciones más sofisticadas en los datos.

A continuación se definirán algunas funciones de activación.

La función sigmoide produce resultados que siempre están en el rango de 0 a 1. Debido a esta propiedad, normaliza la salida de cada neurona en una red neuronal. Se utiliza especialmente en modelos donde necesitamos predecir probabilidades como resultado. Esto se debe a que la probabilidad de cualquier evento siempre está en el rango de 0 a 1, por lo que la función sigmoide es una elección adecuada. Esta función también es conocida por su suave transición entre valores, evitando cambios bruscos en las salidas. Otra ventaja es que la función es diferenciable, lo que significa que podemos calcular la pendiente de la curva sigmoide en cualquier punto. Finalmente, esta función es efectiva para producir predicciones claras que tienden a acercarse a 1 o 0, lo que es útil en tareas de clasificación donde deseamos obtener resultados distintos y bien definidos.

La función "Tanh" es una función tangente hiperbólica. Sus curvas son bastante similares a las de la función sigmoide, pero tiene ventajas sobre la función esta que vale la pena destacar, por ejemplo cuando la entrada es grande o pequeña, la salida es suave y el cambio en la salida es gradual, lo que significa que el gradiente es pequeño. Esto puede dificultar la actualización de los pesos de la red neuronal, lo que es importante para el aprendizaje.

La principal diferencia radica en el rango de valores de salida. La función "tanh" tiene un rango de salida de -1 a 1, y su valor central es 0, lo que es diferente de la función sigmoide. Esto significa que "tanh"

produce salidas que son centradas alrededor de 0 y pueden tener valores negativos.

La ventaja principal de "tanh" es que las entradas negativas se mapean a valores negativos más fuertes, mientras que las entradas cercanas a cero se mapean cerca de cero en el gráfico de la función "tanh". Esto puede ser útil en algunas aplicaciones de redes neuronales donde se desean salidas más balanceadas alrededor de cero.

La función ReLU, o Unidad Rectificadora Lineal, es una función común en las redes neuronales. Lo que hace es muy simple: si la entrada es positiva, deja la entrada sin cambios; si la entrada es negativa, la transforma en cero

Esta función es popular porque es fácil de calcular y ayuda a resolver un problema en el entrenamiento de redes neuronales llamado "desvanecimiento del gradiente". Además, su uso es común en muchas aplicaciones de aprendizaje profundo debido a su eficiencia y efectividad en la práctica.

Para evaluar estos modelos existen distintas métricas, en particular estas nos sirven para cualquier tipo de aprendizaje supervisado.

La métrica precision, es la proporción de verdaderos positivos sobre la suma de falsos positivos y verdaderos negativos. Por ejemplo, en la detección de spam en correos electrónicos, si un correo legítimo se marca como spam por error, es un problema grave. Esto podría resultar en que los usuarios pierdan correos importantes. Así que, en tales casos, es crucial tener un alto nivel de precisión para evitar estos errores costosos.

La métrica accuracy es una medida que nos dice cuán seguido nuestro modelo acierta en sus predicciones. Para calcularla, simplemente dividimos la cantidad de predicciones correctas (todos los verdaderos positivos y verdaderos negativos) entre todas las predicciones realizadas. Es una forma de medir qué tan bien nuestro modelo está haciendo su trabajo.

La métrica Recall evalúa cuántas veces un modelo de aprendizaje automático logra identificar correctamente los casos positivos (verdaderos positivos) de todos los casos positivos reales en el conjunto de datos. Se puede calcular dividiendo el número de verdaderos positivos entre el total de casos positivos.

La matriz de confusión se trata de una métrica de evaluación de rendimiento utilizada en problemas de clasificación en el campo del aprendizaje automático, en los cuales la salida puede pertenecer a dos o más categorías.

III. METODOLOGIA

Lo primero es cargar los datasets

```
!wget https://raw.githubusercontent.com/Felipe1401/Mineria/main/dataset_digits/1_digits_train.txt
```

```
!wget https://raw.githubusercontent.com/Felipe1401/Mineria/main/dataset_digits/1_digits_test.txt
```

Luego, cargamos algunas librerias

```
import pandas as pd
import torch
import torch.nn as nn
import numpy as np
import time
from sklearn.metrics import confusion_matrix,precision_score, recall_score, f1_score, accuracy_score
import seaborn as sn
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

Cargamos los datasets y dataloaders

```
# Crear datasets
feats_train = df_train.to_numpy()[:,0:64].astype(np.float32)
labels_train = df_train.to_numpy()[:,64].astype(int)
dataset_train = [ {"features":feats_train[i,:], "labels":labels_train[i]} for i in range(feats_train.shape[0]) ]

feats_val = df_val.to_numpy()[:,0:64].astype(np.float32)
labels_val = df_val.to_numpy()[:,64].astype(int)
dataset_val = [ {"features":feats_val[i,:], "labels":labels_val[i]} for i in range(feats_val.shape[0]) ]

feats_test = df_test.to_numpy()[:,0:64].astype(np.float32)
labels_test = df_test.to_numpy()[:,64].astype(int)
dataset_test = [ {"features":feats_test[i,:], "labels":labels_test[i]} for i in range(feats_test.shape[0]) ]
```

```
# Crear dataloaders
dataloader_train = torch.utils.data.DataLoader(dataset_train, batch_size=128, shuffle=True, num_workers=0)
dataloader_val = torch.utils.data.DataLoader(dataset_val, batch_size=128, shuffle=True, num_workers=0)
dataloader_test = torch.utils.data.DataLoader(dataset_test, batch_size=128, shuffle=True, num_workers=0)
```

Cambiamos el nombre de las features

```
column_names = ["feat" + str(i) for i in range(64)]
column_names.append("class")
```

```
df_train_val = pd.read_csv('1_digits_train.txt', names = column_names)
df_train_val
```

```
df_test = pd.read_csv('1_digits_test.txt', names = column_names)
df_test
```

Dividimos y normalizamos los datos

```
[73] df_train, df_val = train_test_split(df_train_val, test_size = 0.3, random_state = 10)
```

```
[74] scaler = StandardScaler().fit(df_train.iloc[:,0:64])
     df_train.iloc[:,0:64] = scaler.transform(df_train.iloc[:,0:64])
     df_val.iloc[:,0:64] = scaler.transform(df_val.iloc[:,0:64])
     df_test.iloc[:,0:64] = scaler.transform(df_test.iloc[:,0:64])
```

Función para los modelos

```
def model(capas, activation, neurons):
  if (activation.lower() != "relu" and activation.lower()!= "tanh"):
    return
  if (not (capas <=2 and capas >0)):
    print("Elige entre 1 y dos capas")
    return
  act = nn.ReLU() if activation.lower() == "relu" else nn.Tanh()
  if (capas == 2):
    model = nn.Sequential(
              nn.Linear(64, neurons),
              act,
              nn.Linear(neurons, neurons),
              act,
              nn.Linear(neurons,10)
            )
  else:
    model = nn.Sequential(
              nn.Linear(64, neurons),
              act,
              nn.Linear(neurons, 10),

    )
  return model
```

Función para realizar los plot

```
import matplotlib.pyplot as plt
def plot(params):
  [time, loss_train, loss_val] = params[0], params[1], params[2]
  # Graficar loss de entrenamiento Y validación
  plt.figure(figsize = (8, 5))
  plt.title('Model loss on train & validation')
  plt.xlabel('Time')
  plt.ylabel('Loss')
  plt.plot(time, loss_train, 'b', label = 'Train')
  plt.plot(time, loss_val, 'r', label = 'Validation')
  plt.grid()
  plt.legend()
```

Función para entrenar

```
def train(model):
  device = torch.device('cuda')
  model = model.to(device)
  criterion = nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
  # Guardar resultados del loss y epocas que duró el entrenamiento
  loss_train = []
  loss_val = []
  times = []
  epochs = []
  last = float("Inf")
  # Entrenamiento de la red por n epocas
  start = time.time()
  for epoch in range(1000):

    # Guardar loss de cada batch
    loss_train_batches = []
    loss_val_batches = []

    # Entrenamiento ----------------------------------------------
    model.train()
    true_labels_train = []
    predicted_labels_train = []

    # Debemos recorrer cada batch (lote de los datos)
    for i, data in enumerate(dataloader_train, 0):
      # Procesar batch actual
      inputs = data["features"].to(device) # Características
      labels = data["labels"].to(device)   # Clases
      # zero the parameter gradients
      optimizer.zero_grad()
      # forward + backward + optimize
      outputs = model(inputs)           # Predicciones
      vals, predictions = torch.max(outputs, 1)
      #Guardamos las predicciones del conjunto de entrenamiento
      true_labels_train.extend(labels.cpu().numpy())
      predicted_labels_train.extend(predictions.cpu().numpy())
      loss = criterion(outputs, labels) # Loss de entrenamiento
      loss.backward()                   # Backpropagation
      optimizer.step()

      # Guardamos la pérdida de entrenamiento en el batch actual
      loss_train_batches.append(loss.item())

    # Guardamos el loss de entrenamiento de la época actual
    loss_train.append(np.mean(loss_train_batches)) # Loss promedio de los batches

    # Predicción en conjunto de validación ----------------------------------------
    model.eval()
    true_labels = []
    predicted_labels = []

    with torch.no_grad():
      # Iteramos dataloader_val para evaluar el modelo en los datos de validación
      for i, data in enumerate(dataloader_val, 0):
        # Procesar batch actual
        inputs = data["features"].to(device) # Características
        labels = data["labels"].to(device)   # Clases

        outputs = model(inputs)             # Obtenemos predicciones
        vals, predicted = torch.max(outputs, 1)
        #Guardamos las predicciones del conjunto de validacion
        true_labels.extend(labels.cpu().numpy())
        predicted_labels.extend(predictions.cpu().numpy())
        # Guardamos la pérdida de validación en el batch actual
        loss = criterion(outputs, labels)
        loss_val_batches.append(loss.item())

    # Guardamos el Loss de validación de la época actual
    loss_val.append(np.mean(loss_val_batches)) # Loss promedio de los batches

    # Guardamos la época
    epochs.append(epoch)
    end = time.time()
    times.append(end-start)
    # Imprimir la pérdida de entrenamiento/validación en la época actual
    print(("Epoch: %d, train loss: %.4f, val loss: %.4f"  %(epoch, loss_train[epoch], loss_val[epoch])))
    # Tenemos el loss de entrenamiento y validación, ¿Como sería el early-stopping?
    if (loss_val[epoch] > last):
      break
    else:
      last = loss_val[epoch]

  return [times, loss_train, loss_val, true_labels, predicted_labels, true_labels_train, predicted_labels_train]
```

Función para calcular las métricas de los modelos

```
def metrics(y_pred, y_true):
  precision = precision_score(y_true, y_pred, average='weighted')
  recall = recall_score(y_true, y_pred, average='weighted')
  f1 = f1_score(y_true, y_pred, average='weighted')
  accuracy = accuracy_score(y_true, y_pred, normalize = True)
  return[accuracy, precision, recall, f1]
```

Una vez hecho todo esto, se procede a entrenar cada modelo. Se entrena y se plotea la Perdida en función del tiempo transcurrido.

```
md = model(1, "relu", 10)
params = train(md)
plot(params)
```

```
md = model(1, "relu", 40)
params = train(md)
plot(params)
```

```
md = model(1, "tanh", 10)
params = train(md)
plot(params)
```

```
md = model(1, "tanh", 40)
params = train(md)
plot(params)
```

```
md = model(2, "relu", 10)
params = train(md)
plot(params)
```

```
md = model(2, "relu", 40)
params = train(md)
plot(params)
```

Para cada uno de los modelos, se calcula la matriz de confusion, tanto para el conjunto de validación

como entrenamiento, además de las métricas asociadas al conjunto de validación.

```
true_labels_train, predicted_labels_train = params[5], params[6]
confusion = confusion_matrix(true_labels_train, predicted_labels_train, normalize= "true")
plt.figure(figsize = (12,7))
plt.title("Matriz de confusión conjunto de entrenamiento")
sn.heatmap(confusion, annot=True)
```

```
true_labels_val, predicted_labels_val = params[3], params[4]
confusion = confusion_matrix(true_labels_val, predicted_labels_val, normalize= "true")
plt.figure(figsize = (12,7))
plt.title("Matriz de confusión conjunto de validación")
sn.heatmap(confusion, annot=True)
```

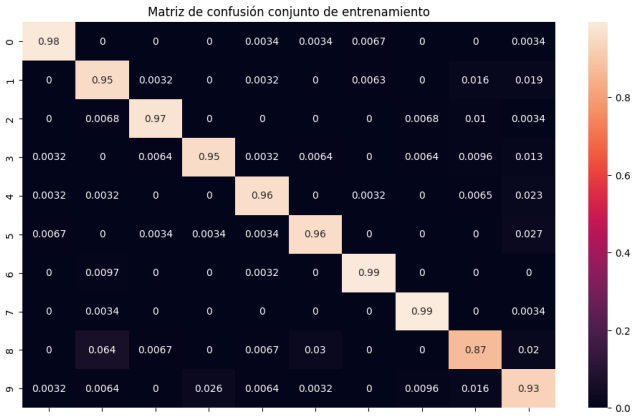
```
[accuracy, precision, recall, f1] = metrics(true_labels_val, predicted_labels_val)
print("Métricas para el conjunto de validación")
print(f"Accuracy : {accuracy}")
print(f"Precision : {precision}")
print(f"Recall : {recall}")
print(f"f1_score : {f1}")
```

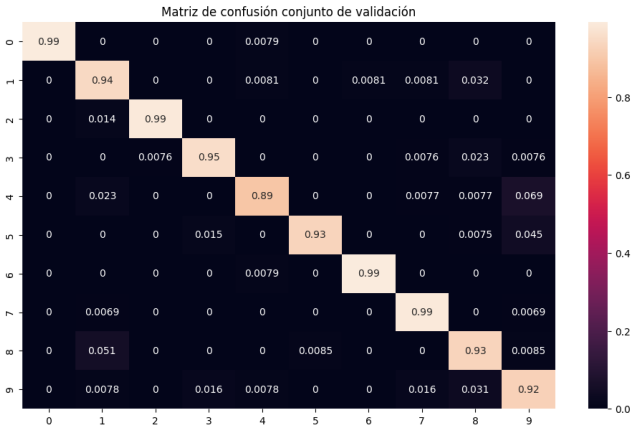
Finalmente, debemos encontrar el mejor modelo y calcular la matriz de confusion tal como se hizo anteriormente, pero para el conjunto de testeo de este, además de las métricas asociadas. Para esto, modificamos la función train de tal forma que podamos obtener los valores de este conjunto, añadiendo el siguiente bloque de código.

```
true_labels_test = []
predicted_labels_test = []

with torch.no_grad():
    for i, data in enumerate(dataloader_test, 0):
        # Procesa el lote actual
        inputs = data["features"].to(device)  # Características
        labels = data["labels"].to(device)    # Clases

        outputs = model(inputs)               # Obtiene predicciones
        vals, predictions = torch.max(outputs, 1)
        # Guarda las predicciones del conjunto de prueba
        true_labels_test.extend(labels.cpu().numpy())
        predicted_labels_test.extend(predictions.cpu().numpy())
```

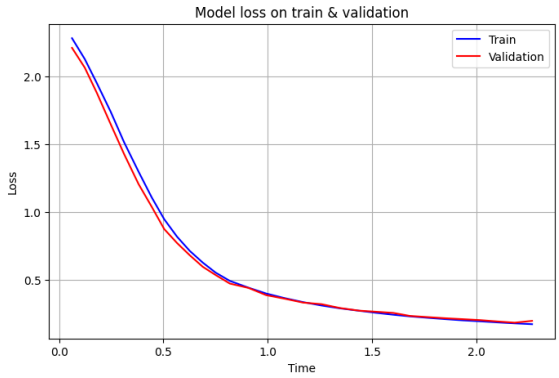




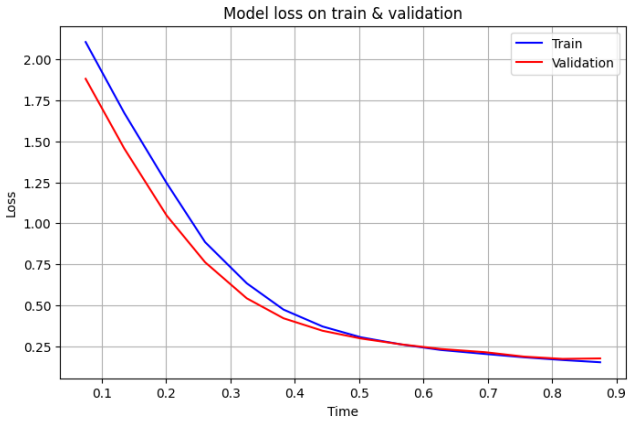

```
Métricas para el conjunto de validación
Accuracy : 0.9540229885057471
Precision : 0.9543212495737378
Recall : 0.9540229885057471
f1_score : 0.9537530131903138
```

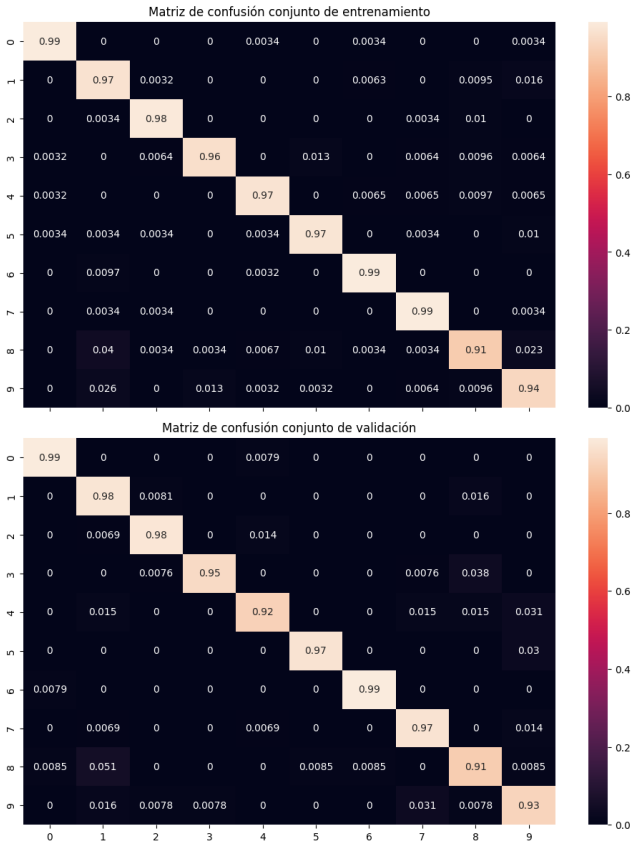
IV. Resultados

a) 10 neuronas en la capa oculta, usando como función de activación ReLU y 1000 épocas como máximo

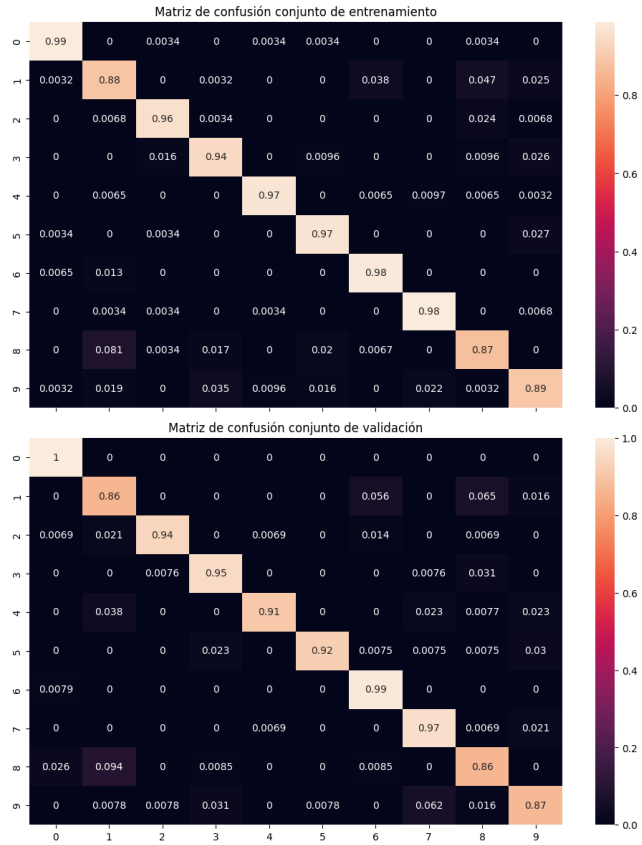


b) 40 neuronas en la capa oculta, usando como función de activación ReLU y 1000 épocas como máximo



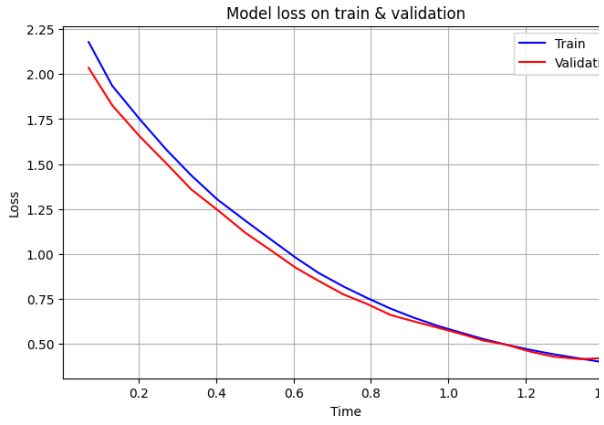



```
Métricas para el conjunto de validación
Accuracy : 0.9601532567049809
Precision : 0.9604780770453525
Recall : 0.9601532567049809
f1 score : 0.9600654935370007
```

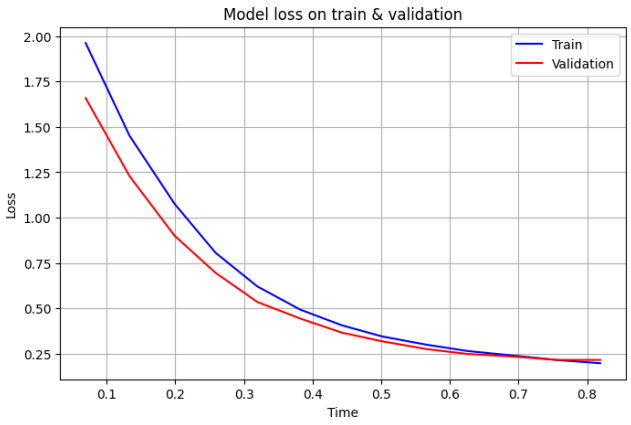


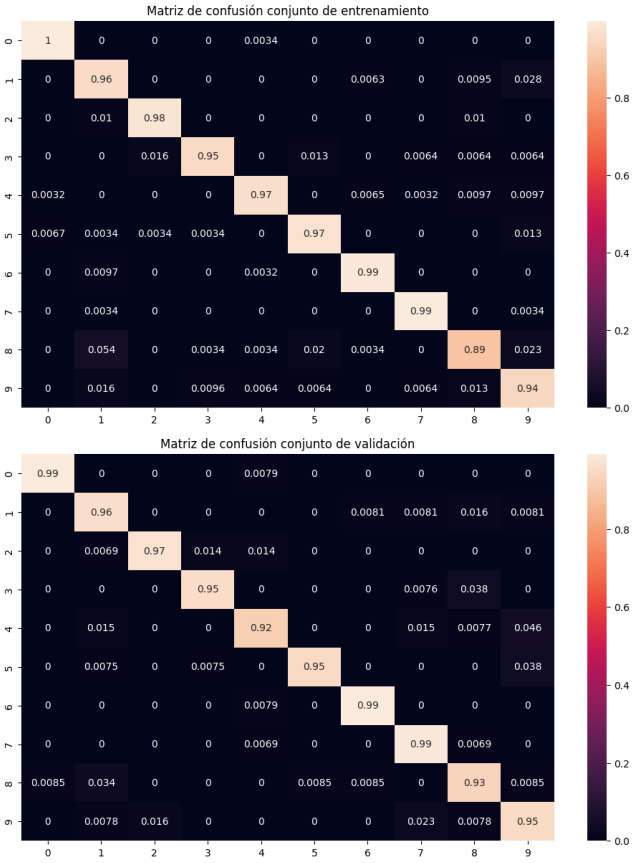

```
Métricas para el conjunto de validación
Accuracy : 0.9295019157088122
Precision : 0.9306195219982781
Recall : 0.9295019157088122
f1 score : 0.9294458422872445
```

c) 10 neuronas en la capa oculta, usando como función de activación Tanh y 1000 épocas como máximo.



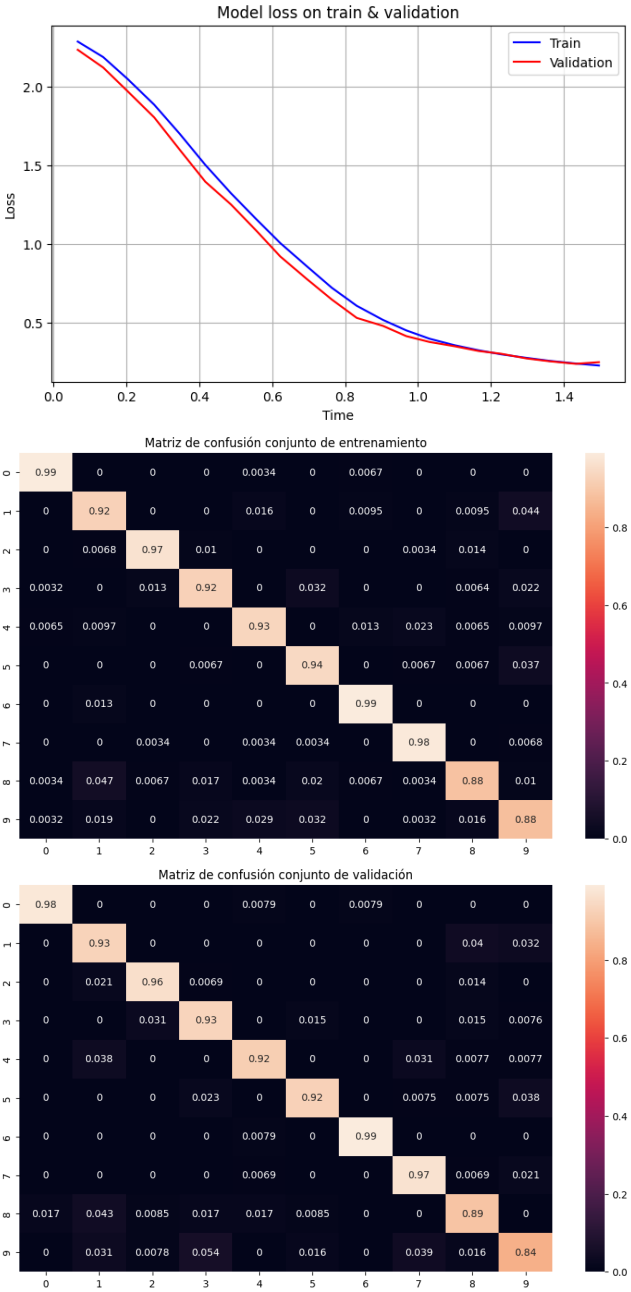
d) 40 neuronas en la capa oculta, usando como función de activación Tanh y 1000 épocas como máximo.





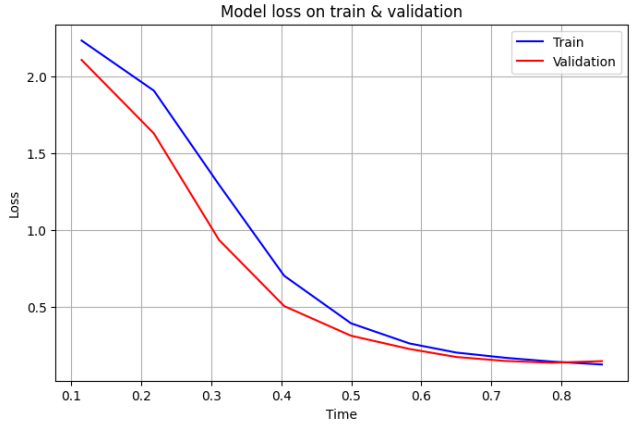

```
Métricas para el conjunto de validación
Accuracy : 0.9593869731800766
Precision : 0.9596659373342018
Recall : 0.9593869731800766
f1_score : 0.9592895586514896
```

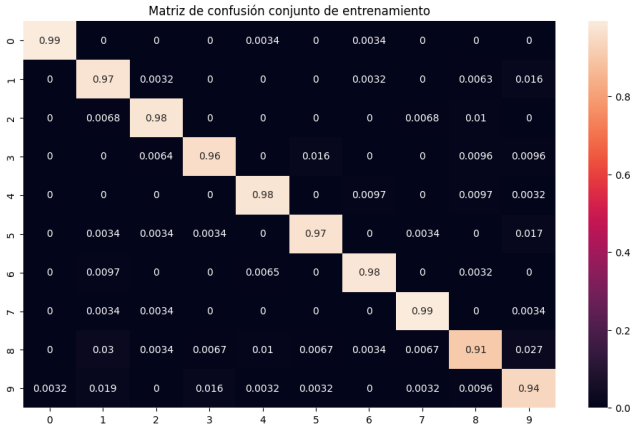
e) 2 capas ocultas con 10 y 10 neuronas cada una y función de activación ReLU, y 1000 épocas como máximo.

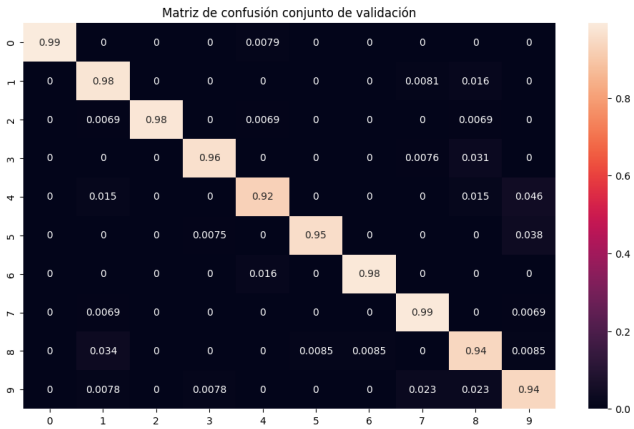



```
Métricas para el conjunto de validación
Accuracy : 0.9333333333333333
Precision : 0.9340099544039996
Recall : 0.9333333333333333
f1_score : 0.9333820733005129
```

f) 2 capas ocultas con 40 y 40 neuronas cada una y función de activación ReLU, y 1000 épocas como máximo.








```
Métricas para el conjunto de validación
Accuracy : 0.963984674329502
Precision : 0.9640550210300487
Recall : 0.963984674329502
f1_score : 0.9637705036276905
```

V. ANÁLISIS DE LOS RESULTADOS

En general, todos los modelos con distintas capas y distintas funciones de activación tuvieron un muy buen rendimiento. Todos tuvieron accuracy, recall, f1 score y precision mayor a 0.9, de hecho, quien obtuvo las métricas más bajas fue el modelo de 10 neuronas en la capa oculta, usando como función de activación Tanh y 1000 épocas como máximo, obteniendo valores de al rededor de 0.929, lo cual de igual manera es super bueno.

Las matrices de confusion son bien parecidas en todos los casos, un caso repetido es que los verdaderos positivos de la clase “9”, son quienes tienen menor valor en general. Por otro lado, tanto las matrices en el conjunto de entrenamiento como en el de validación tienen valores similares, esto hace que el modelo sea confinable y tenga una generalización a cualquier input (evitamos el overfitting).

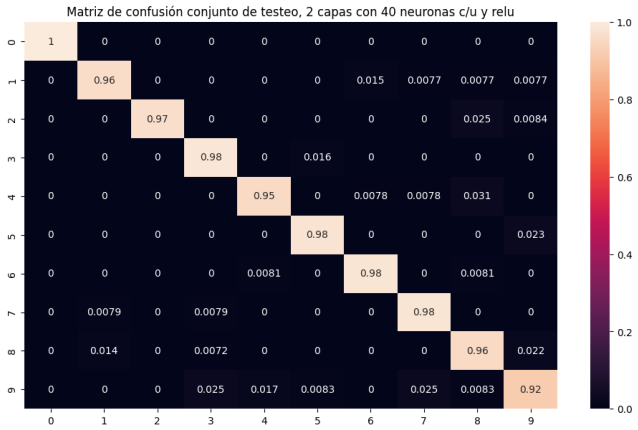
El efecto de varias neuronas en este caso arroja mayores valores en las métricas, por ejemplo pasar de 10 neuronas a 40 tanto en ReLu como en Tanh hace que el accuracy sea mayor. Lo mismo al aumentar las capas, pasar de una a dos capas también hace que las métricas aumenten. En cuanto a las funciones de activación.

Por otro lado, al comparar las funciones de activación, los modelos que utilizan ReLu son quienes obtuvieron mejores resultados.

Independientemente de las capas, al aumentar las neuronas, el tiempo en que se termina de entrenar la red, es decir cuando se active el early stopping, es menor cuando hay mas neuronas.

Por otro lado, el mejor resultado, fue el modelo con 2 capas y 40 neuronas cada una, usando la función de activación ReLu. Este modelo obtuvo valores del orden de 0.963 aproximadamente, un valor demasiado cerca de 1.

A continuación, veremos que tal anda este “mejor modeo” en el conjunto de test (prueba), graficando la matriz de confusion y las métricas asociadas.




```
Métricas para el conjunto de testeo
Accuracy : 0.9685534591194969
Precision : 0.9687368510434099
Recall : 0.9685534591194969
f1_score : 0.9685439887478585
```

Como era de esperarse, también anda super bien, obteniendo métricas cercanas a 0.97, si comparamos con respecto al conjunto de validación y entrenamiento, la matriz de confusion nos muestra que hay más verdaderos positivos y verdaderos negativos para las clases en general, es mínimo, pero existe.

VI. CONCLUSIONES

Además de tener separados los conjuntos de entrenamiento y testeo, es importante tener un conjunto de validación, pues al entrenar las redes se tenía como máximo 1000 epochs, pero no fue necesario pasar por todas, de hecho ni se acercó a este valor debido al early stopping.

La función ReLu da mejores resultados que la función tangente hiperbólica.

El usar más capas y más neuronas nos permite que la red neuronal alcance el early stopping de una forma más rápida, lo cual implica menor tiempo de entrenamiento.

Los gráficos y la evaluación de las métricas para distintos modelos son muy importantes para tomar la decision de cuál modelo es mejor para cierto problema a resolver con métodos de aprendizaje supervisado como lo son las Redes Neuronales.

REFERENCES

[1] https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[2] https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

[3] https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6