Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Tarea 4

Redes Neuronales Artificiales

Integrantes: Bastián Garcés
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla Z.
Ayudantes: Juan Pablo Cáceres B.
Rudy García
Rodrigo Salas O.
Sebastian Solanich
Pablo Troncoso P.

Fecha de entrega: 6 de Junio de 2021
Santiago, Chile

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Introducción

En este informe se detallará la tarea número 4 del curso EL4106 Inteligencia Computacional, tarea que consiste en implementar redes neuronales que tendrán 1 o 2 capas ocultas y usarán funciones de activación ReLU o tangente hiperbólica, para que posteriormente se obtenga el accuracy y la matriz de confusión para las distintas redes a implementar. Para la realización de esta tarea se recurrirá a un subconjunto de la base de datos *Dataset for Sensorless Drive Diagnosis Data Set*, que corresponde a una base de datos tomada del *UC Irvine Machine Learning Repository*, dicha base de datos consiste en 48 características que fueron extraídas de la señal de corriente que alimenta a un motor síncrono, existiendo 11 clases. También se dispondrá de un código base entregado por el equipo docente, en dicho código se tienen las configuraciones iniciales de la red neuronal, como la normalización de características, creación de la red neuronal, creación de dataloaders entre otras. Además se recurrirá a las librerías *Pandas*, *Numpy*, *Torch*, *Seaborn*, *Time*, *Matplotlib*, *Sklearn* y *Sys*. Las librerías mencionadas son utilizadas en el lenguaje de programación *Python* y el entorno de trabajo a utilizar corresponde a *Colaboratory*.

Las secciones que tendrá el presente informe se corresponden con las distintas actividades que el cuerpo docente solicitó realizar. A continuación se enlistan las distintas secciones que tendrá el documento:

- **Actividad 1:** Esta actividad consiste únicamente en ejecutar el código base entregado por el cuerpo docente e indicar si este se ejecutó sin problemas.
- **Actividad 2:** En la segunda actividad se pide entrenar una red neuronal que tenga las siguientes características:
  - **a)** 10 neuronas en la capa oculta, utilizando una función de activación ReLU y 1000 épocas.
  - **b)** 40 neuronas en la capa oculta, utilizando una función de activación ReLU y 1000 épocas.
  - **c)** 10 neuronas en la capa oculta, utilizando una función de activación Tanh y 1000 épocas.
  - **d)** 40 neuronas en la capa oculta, utilizando una función de activación Tanh y 1000 épocas.
  - **e)** 2 capas ocultas, ambas con 10 neuronas, usando una función de activación ReLU y 1000 épocas.
  - **f)** 2 capas ocultas, ambas con 40 neuronas, usando una función de activación ReLU y 1000 épocas.

  La dificultad de esta actividad recae principalmente en la forma en que se impedirá el sobreajuste de la red neuronal. Para evitar dicho sobreajuste el estudiante deberá implementar un código que sea capaz de detectar el momento en que la red se esté sobreajustando y detenga el entrenamiento. Luego, para cada red entrenada, se pide que se calcule el loss de entrenamiento y de validación, así como también el tiempo de procesamiento requerido, para posteriormente realizar un gráfico en el que se vean ambos loss en función de las épocas. Posteriormente se pide obtener el accuracy y matriz de confusión usando el conjunto de entrenamiento y validación.
- **Actividad 3:** En esta actividad se solicita obtener el accuracy y matriz de confusión de la mejor red obtenida en validación, cabe destacar que la red a utilizar será aquella que tenga mayor accuracy al utilizar el conjunto de validación.

- **Actividad 4:** La última actividad consiste en realizar un análisis de los resultados obtenidos durante las actividades previas, se solicita:
  - **a)** Explicar el efecto de cambiar la cantidad de neuronas en la capa oculta y como esto afecta al desempeño de la red.
  - **b)** Explicar como afecta el variar la cantidad de capas ocultas y como esta variación afecta el desempeño de la red.
  - **c)** Explicar los efectos de utilizar las distintas funciones de activación y como el uso de estas afecta a la red en su desempeño.
  - **d)** Analizar tiempos de entrenamiento, matrices de confusión y accuracies de las distintas redes probadas en el conjunto de validación.
  - **e)** Analizar la matriz de confusión y el accuracy en el conjunto de prueba, respecto a los obtenidos al utilizar el conjunto de validación.

2. Desarrollo

A continuación se enunciarán las diversas actividades a desarrollar durante la tarea, cabe destacar que a lo largo de esta sección no se explicará que es lo que se debe desarrollar en cada actividad, esto ya se explicó en la introducción.

2.1. Actividad 1

Como se mencionó en la introducción esta actividad no tenía mayor complejidad que correr el código base entregado por el cuerpo docente, dicho código es el que se puede observar en el código 1. Dicho código no presentó problemas en su ejecución.

Código 1: Código base.

```
import pandas as pd
import torch
import torch.nn as nn

import numpy as np

from torch.utils.data import random_split

import sys

#----------------------

from google.colab import files
uploaded = files.upload()  # sensorless_tarea4_train.txt

from google.colab import files
uploaded = files.upload()  # sensorless_tarea4_test.txt

#----------------------

!ls

column_names = ["feat" + str(i) for i in range(48)]
column_names.append("class")

df_train_val = pd.read_csv('sensorless_tarea4_train.txt', names = column_names)
df_train_val["class"] = df_train_val["class"] - 1
df_train_val

df_test = pd.read_csv('sensorless_tarea4_test.txt', names = column_names)
df_test["class"] = df_test["class"] - 1
df_test

#----------------------

model = nn.Sequential(
```

```
        nn.Linear(48, 11)
      )

device = torch.device('cuda')

model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

#----------------------

# Crear datasets

feats_train_val = df_train_val.to_numpy()[:,0:48].astype(np.float32)
labels_train_val = df_train_val.to_numpy()[:,48].astype(int)
dataset_train_val = [ {"features":feats_train_val[i,:], "labels":labels_train_val[i]} for i in range(
    ↪ feats_train_val.shape[0]) ]

feats_test = df_test.to_numpy()[:,0:48].astype(np.float32)
labels_test = df_test.to_numpy()[:,48].astype(int)
dataset_test = [ {"features":feats_test[i,:], "labels":labels_test[i]} for i in range(feats_test.shape[0]) ]

n_train = int(df_train_val.shape[0]*0.85)
n_val = df_train_val.shape[0] - n_train

dataset_train, dataset_val = random_split(dataset_train_val, [n_train, n_val])


# Normalizar datos

fdata = []
i = 0

for x in dataset_train:
  fdata.append(x['features'])

fdata = np.array(fdata)

fmean= np.mean(fdata, axis=0)
fstd = np.std(fdata, axis=0)

for x in dataset_train:
  x['features'] = (x['features']-fmean) / fstd

for x in dataset_val:
  x['features'] = (x['features']-fmean) / fstd

for x in dataset_test:
  x['features'] = (x['features']-fmean) / fstd

```

```
87
88 # Crear dataloaders
89 dataloader_train = torch.utils.data.DataLoader(dataset_train, batch_size=128, shuffle=True,
       ↪ num_workers=0)
90 dataloader_val = torch.utils.data.DataLoader(dataset_val, batch_size=128, shuffle=True,
       ↪ num_workers=0)
91 dataloader_test = torch.utils.data.DataLoader(dataset_test, batch_size=128, shuffle=True,
       ↪ num_workers=0)
92
93 #----------------------
94
95 for epoch in range(1000):
96     try:
97       sys.stdout.write("\rÉpoca %d  "  %(epoch))
98       model.train()
99
100      # Train on the current epoch
101      for i, data in enumerate(dataloader_train, 0):
102          inputs = data["features"].to(device)
103          labels = data["labels"].to(device)
104
105          # zero the parameter gradients
106          optimizer.zero_grad()
107
108          # forward + backward + optimize
109          outputs = model(inputs)
110
111          loss = criterion(outputs, labels)
112          loss.backward()
113          optimizer.step()
114
115      # Compute validation loss and accuracy for current epoch
116      model.eval()
117
118      with torch.no_grad():
119        for i, data in enumerate(dataloader_val, 0):
120            inputs = data["features"].to(device)
121            labels = data["labels"].to(device)
122
123            outputs = model(inputs)
124            # Calcular loss de validación
125
126      # Imprimir: numero de época, loss de entrenamiento y loss de validación
127      # Se debe usar sys.stdout.write() para que la línea de texto se sobreescriba en vez de imprimirse
       ↪  línea por línea
128      # No se debe guardar checkpoints en cada época (guardarlos cada 50 épocas)
129
130    except KeyboardInterrupt:
131      print("\nEntrenamiento interrumpido")
132      break
133
```

```
134 print('\nEntrenamiento finalizado')
```

2.2. Actividad 2

En esta actividad se explicará como tal el entrenamiento diseñado y como este decide cuando es tiempo de detenerse. El entrenamiento diseñado corresponde al del código 2.

Código 2: Entrenamiento.

```
tiempo_ini = time.time()

prom_loss_train = [] #Promedio de loss del entrenamiento para cada epoca
prom_loss_vali = [] #Promedio del loss de validacion para cada epoca
epocas = range(49,1000,50)
prom_loss_train_def = [] #Promedio de loss cada 50 epocas (Entrenamiento)
prom_loss_vali_def = [] #Promedio loss cada 50 epocas (Validacion)
for epoch in range(1000):
    try:
      model.train()

      # Train on the current epoch
      loss_epoch_train = [] #Acá se guarda el loss de cada dato de la epoca
      for i, data in enumerate(dataloader_train, 0):
          inputs = data["features"].to(device)
          labels = data["labels"].to(device)

          # zero the parameter gradients
          optimizer.zero_grad()

          # forward + backward + optimize
          outputs = model(inputs)

          loss_train = criterion(outputs, labels) #calculo del loss de entrenamiento por dato
          loss_epoch_train.append(loss_train)     #guardamos el loss del dato en un vector
          loss_train.backward()
          optimizer.step()
      prom_train = sum(loss_epoch_train)/len(loss_epoch_train) #Calculamos el loss promedio de
      ↪ la epoca
      prom_loss_train.append(prom_train) #Guardamos el promedio en un vector

      # Compute validation loss and accuracy for current epoch
      model.eval()

      with torch.no_grad():
        loss_epoch_vali = [] #Acá se guarda el loss de cada dato de la epoca
        for i, data in enumerate(dataloader_val, 0):
            inputs = data["features"].to(device)
            labels = data["labels"].to(device)

            outputs = model(inputs)
```

```
41          # Calcular loss de validación
42          loss_vali = criterion(outputs, labels) #calculo del loss de validación por dato
43          loss_epoch_vali.append(loss_vali)      #guargamos el loss del dato en un vector
44        prom_vali = sum(loss_epoch_vali)/len(loss_epoch_vali) #Calculamos el loss promedio de la
      ↪ epoca
45        prom_loss_vali.append(prom_vali)
46
47      # Imprimir: numero de época, loss de entrenamiento y loss de validación
48      # Se debe usar sys.stdout.write() para que la línea de texto se sobreescriba en vez de imprimirse
      ↪  línea por línea
49      # No se debe guardar checkpoints en cada época (guardarlos cada 50 épocas)
50
51      sys.stdout.write("\rÉpoca %d, loss entrenamiento %f, loss validacion %f" % (epoch,
      ↪ prom_train, prom_vali))
52
53      #Hacemos el checkpoint cada 50 epocas y revisamos si debemos interrumpir el entrenamiento
54      if epoch in epocas:
55        torch.save(model, 'red'+str(epocas.index(epoch))+'.pt')
56        s_train = 0 #Suma de los loss de las 50 epocas anteriores (entrenamiento)
57        s_vali = 0 #Suma de los loss de las 50 epocas anteriores (validacion)
58        for i in range(epoch-50,epoch):
59          s_train = prom_loss_train[i]+s_train
60          s_vali = prom_loss_vali[i]+s_vali
61        prome_train = s_train/50 #Loss promedio de las 50 epocas (Entrenamiento)
62        prome_vali = s_vali/50 #Loss promedio de las 50 epocas (Validacion)
63        prom_loss_train_def.append(prome_train)
64        prom_loss_vali_def.append(prome_vali)
65        #Si el loss de validacion actual es mayor que el de la vez pasada termino
66        #El entrenamiento
67        if prom_loss_vali_def[len(prom_loss_vali_def)-1] > prom_loss_vali_def[len(
      ↪ prom_loss_vali_def)-2]:
68          print('\nEl loss de validacion empezo a aumentar en la epoca '+str(epoch))
69          print('Por lo tanto nos quedaremos con la red de la epoca '+str(epoch-50))
70          model = torch.load('red'+str(epocas.index(epoch)-1)+'.pt') #Nos quedamos con la red
      ↪ anterior
71          break
72
73    except KeyboardInterrupt:
74      print("\nEntrenamiento interrumpido")
75      break
76
77  print('\nEntrenamiento finalizado')
78  tiempo_fin = time.time()
79  print("El entrenamiento tardó: "+str(tiempo_fin-tiempo_ini))
```

A continuación se explicarán las diversas variables del código así como también la condición para que este se detenga. Primero se define un tiempo, este tiempo consiste en el tiempo en que el entrenamiento comienza, luego se definen vectores, el primer par de vectores observados se encargan de almacenar el loss promedio por época, un vector guarda el loss promedio del entrenamiento en la época y el otro el loss promedio del conjunto de validación en la época. Posteriormente se define un vector que corresponde a las épocas en que el algoritmo tendrá que chequear si se debe detener el

entrenamiento, este vector va desde 49 a 1000 dando pasos de 50, es decir, en la época 49 (correspondiente a la época 50 pues *Python* considera la época 0 como la época 1) se revisa por primera vez si se debe detener el entrenamiento, luego se vuelve a revisar en la época 99, 149 y así sucesivamente, hasta que la condición de "detención" del entrenamiento se cumpla. Finalmente se tiene otro par de vectores, estos almacenan el loss promedio cada 50 épocas, debido a lo anterior guardarán un valor cada 50 épocas, para clarificar se enunciará un ejemplo. Supongamos que el código va iterando en la época 199, al ser una época en que se debe revisar si se debe detener el entrenamiento se calcula el loss promedio de las 50 épocas anteriores, es decir, los loss promedio almacenados en los vectores "prom_loss_train" y "prom_loss_vali" desde la celda 149 a la 199 se promedian y el valor obtenido se guarda en los vectores "prom_loss_train‘_def" y "prom_loss_vali_def" según corresponda.

Con lo anterior mencionado se procederá a explicar el entrenamiento como tal. Al momento de comenzar a entrenar en cada época se define otro vector, vector que se encargará de guardar el loss del dato que se está revisando, luego, una vez se han revisado todos los datos, se calcula el promedio de todos los loss del conjunto de entrenamiento (loss que están guardados en el vector "loss_epoch_train") y el valor obtenido se añade al vector "prom_loss_train". Luego al momento de evaluar el modelo implementado en el conjunto de validación se procede a repetir exactamente lo mismo, se define un vector en el que se guardará el loss de cada dato y luego se calculará el promedio de ese vector y el resultado se añadirá al vector "prom_loss_vali". Una vez se entrenó el modelo y se evaluó en el conjunto de validación simplemente se muestran en pantalla la época actual, el loss promedio de la época en el conjunto de entrenamiento y el loss promedio de la época en el conjunto de validación, donde se utilizó la función *sys.stdout.write()* para que la línea se vaya sobrescribiendo y no se muestre una para cada época. Finalmente se tiene una condición, esta corresponde a que si el código está iterando en una época múltiplo de 50 se debe chequear si detener el entrenamiento, dentro de esta condición se guarda el modelo que está implementado hasta el momento, para esto se hace uso de la función *torch.save()*, la finalidad de guardar los modelos implementados es la de dejar un "checkpoint" de la red implementada al momento de revisar si detener el entrenamiento para así recuperarla en caso de que sea necesario, luego simplemente se calcula el promedio de los promedios de los loss de entrenamiento y validación y se añaden a un vector, llegado a ese punto de la ejecución del código se debe revisar si el loss promedio de validación actual es mayor que el de las 50 épocas anteriores, si esto pasa se debe detener el entrenamiento, motivo por el cual se recupera el modelo guardado en las 50 épocas anteriores (momento en que el loss fue mínimo) y se rompe el bucle, de forma tal que el entrenamiento se detenga. Finalmente se obtiene el tiempo en que el entrenamiento finalizó y se muestra en pantalla cuanto tiempo tardó en entrenarse la red hasta que el loss de validación comenzó a aumentar.

A continuación se expondrá la implementación de los 6 tipos de redes presentados en la introducción así como también las distintas métricas de estas. Cabe destacar que los gráficos de loss solo llegan hasta la época en que el loss de validación fue mínimo, si el entrenamiento termina en la época 299 la red a utilizar será la guardada en la época 249 y el loss será graficado hasta esa época, además, se debe tener en cuenta que el código implementado solo implementa 1 red, por lo cual si se desea cambiar el tipo de red a utilizar se debe redefinir el modelo y ejecutar el código desde 0 idealmente.

- **10 neuronas en capa oculta, utilizando función de activación ReLU y 1000 epocas:** El código donde se aprecia la implementación de la red neuronal en cuestión se observa en el código 3.

Código 3: Primera red neuronal.

```
model = nn.Sequential(
  nn.Linear(48, 10),
  nn.ReLU(),
  nn.Linear(10,11)
)

```

Para la red neuronal anterior se obtuvo el tiempo de entrenamiento observado en el código 4.

Código 4: Tiempo de entrenamiento de la primera red.

```
Época 399, loss entrenamiento 0.102487, loss validacion 0.181362
El loss de validacion empezo a aumentar en la epoca 399
Por lo tanto nos quedaremos con la red de la epoca 349

Entrenamiento finalizado
El entrenamiento tardó: 18.186123371124268

```

Luego el gráfico de loss de entrenamiento y validación por epoca se puede observar en la figura 1.

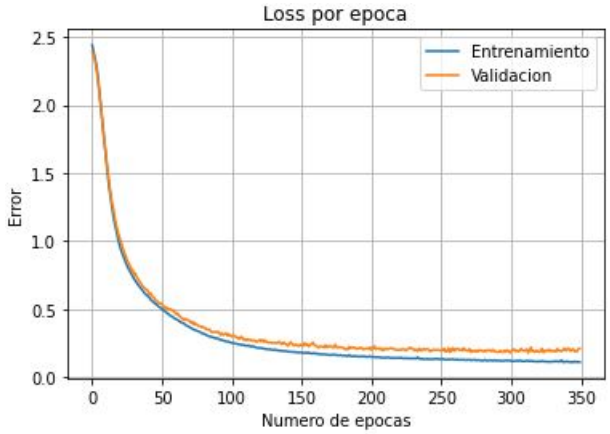

Figura 1: Loss por época para la primera red.

El accuracy de la red en el conjunto de entrenamiento se observa en el código 5 y la matriz de confusión es la que se observa en la figura 2.

Código 5: Accuracy de la primera red sobre el conjunto de entrenamiento.

```
El accuracy de la red al usar el conjunto de entrenamiento es: 0.9641089108910891

```

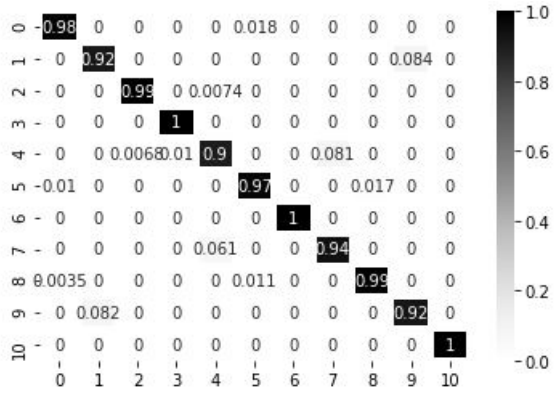

Figura 2: Matriz de confusión de la primera red sobre el conjunto de entrenamiento.

Finalmente el accuracy y la matriz de confusión de la red sobre el conjunto de validación son los correspondientes al código 6 y la figura 3.

Código 6: Accuracy de la primera red sobre el conjunto de validación.

```
El accuracy de la red al usar el conjunto de validacion es: 0.9352014010507881
```

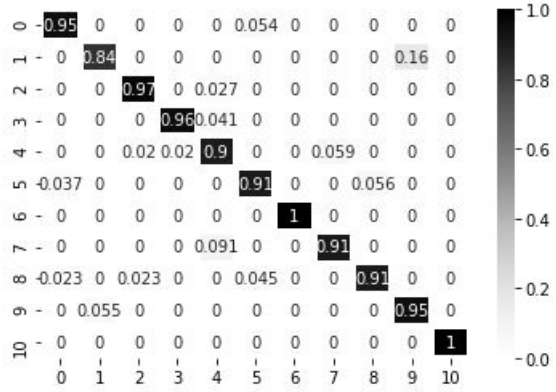

Figura 3: Matriz de confusión de la primera red sobre el conjunto de validación.

- **40 neuronas en capa oculta, utilizando función de activación ReLU y 1000 epocas:** El código donde se aprecia la implementación de la red neuronal se observa en el código 7.

Código 7: Segunda red neuronal.

```
model = nn.Sequential(
  nn.Linear(48, 40),
  nn.ReLU(),
  nn.Linear(40,11)
)
```

Para la red neuronal implementada se obtuvo el tiempo de entrenamiento observado en el código 8.

Código 8: Tiempo de entrenamiento de la segunda red.

```
Época 249, loss entrenamiento 0.013188, loss validacion 0.176673
El loss de validacion empezo a aumentar en la epoca 249
Por lo tanto nos quedaremos con la red de la epoca 199

Entrenamiento finalizado
El entrenamiento tardó: 11.357174396514893

```

Posteriormente, al graficar loss de entrenamiento y validación por época se obtuvo el gráfico de la figura 4.

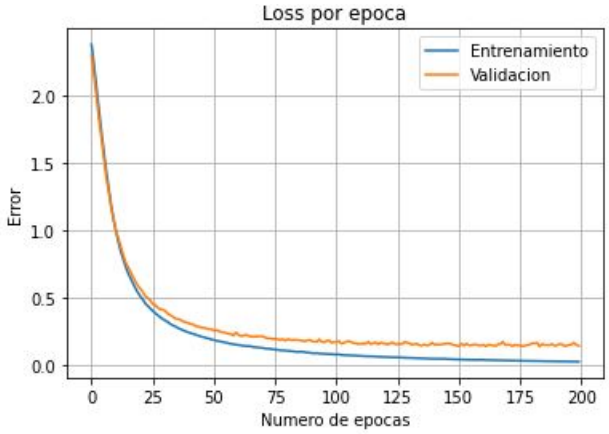

Figura 4: Loss por época para la segunda red.

El accuracy de la red sobre el conjunto de entrenamiento se observa en el código 9 y la matriz de confusión es la que se aprecia en la figura 5.

Código 9: Accuracy de la segunda red sobre el conjunto de entrenamiento.

```
El accuracy de la red al usar el conjunto de entrenamiento es: 0.9984529702970297

```

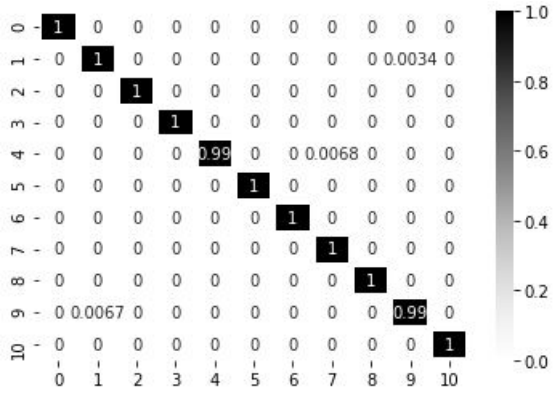

Figura 5: Matriz de confusión de la segunda red sobre el conjunto de entrenamiento.

Finalmente el accuracy y la matriz de confusión de la red aplicada sobre el conjunto de validación son los correspondientes al código 10 y la figura 6.

Código 10: Accuracy de la segunda red sobre el conjunto de validación.

```
El accuracy de la red al usar el conjunto de validacion es: 0.9439579684763573
```

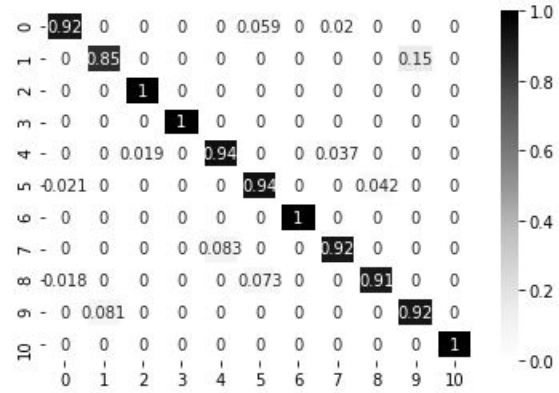

Figura 6: Matriz de confusión de la segunda red sobre el conjunto de validación.

- **10 neuronas en capa oculta, utilizando función de activación Tanh y 1000 épocas:** El código donde se aprecia la implementación de la red neuronal se observa en el código 11.

Código 11: Tercera red neuronal.

```
model = nn.Sequential(
  nn.Linear(48, 10),
  nn.Tanh(),
  nn.Linear(10,11)
)
```

Para la red neuronal apreciada en el código anterior se obtuvo el tiempo de entrenamiento observado en el código 12.

Código 12: Tiempo de entrenamiento de la tercera red.

```
Época 949, loss entrenamiento 0.089275, loss validacion 0.163774
El loss de validacion empezo a aumentar en la epoca 949
Por lo tanto nos quedaremos con la red de la epoca 899

Entrenamiento finalizado
El entrenamiento tardó: 42.90746855735779

```

Posteriormente, al graficar loss de entrenamiento y validación por época se obtuvo el gráfico de la figura 7.

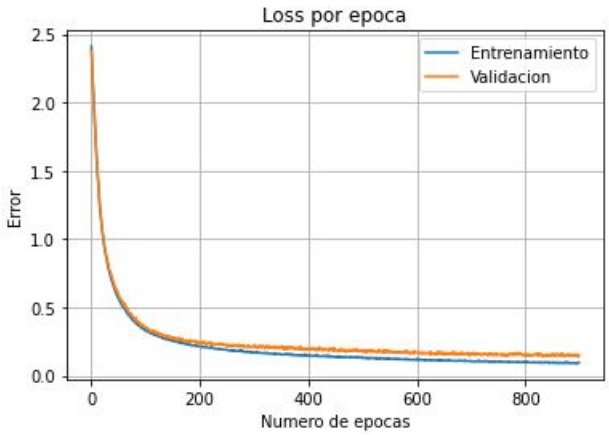

Figura 7: Loss por época para la tercera red.

Al aplicar la red sobre el conjunto de entrenamiento se obtuvo el accuracy apreciado en el código 13 y la matriz de confusión correspondiente es la que se aprecia en la figura 8.

Código 13: Accuracy de la tercera red sobre el conjunto de entrenamiento.

```
El accuracy de la red al usar el conjunto de entrenamiento es: 0.973700495049505

```

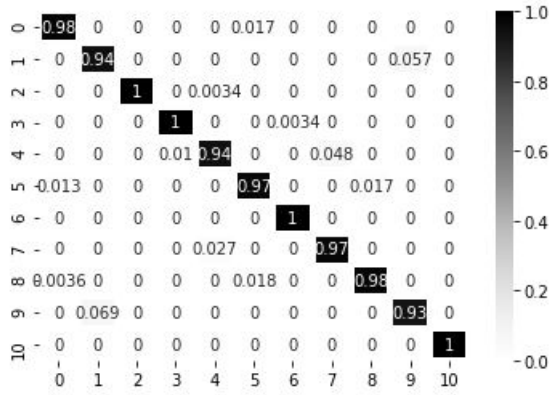

Figura 8: Matriz de confusión de la tercera red sobre el conjunto de entrenamiento.

Finalmente el accuracy y la matriz de confusión de la red al aplicarla sobre el conjunto de validación son los correspondientes al código 14 y la figura 9.

Código 14: Accuracy de la tercera red sobre el conjunto de validación.

```
El accuracy de la red al usar el conjunto de validacion es: 0.9439579684763573
```

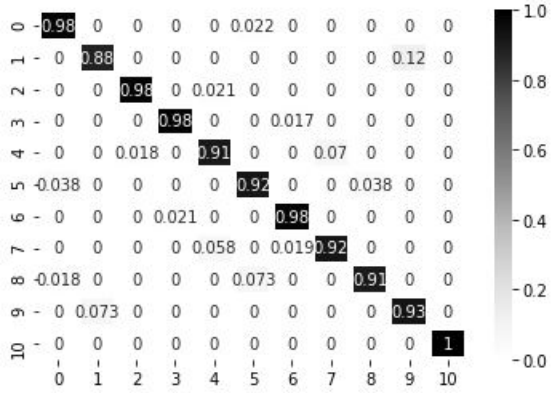

Figura 9: Matriz de confusión de la tercera red sobre el conjunto de validación.

- **40 neuronas en capa oculta, utilizando función de activación Tanh y 1000 épocas:** La red neuronal implementada se puede observar en el código 15.

Código 15: Cuarta red neuronal.

```
model = nn.Sequential(
  nn.Linear(48, 40),
  nn.Tanh(),
  nn.Linear(40,11)
)
```

La red neuronal en cuestión tuvo el tiempo de entrenamiento observado en el código 16.

Código 16: Tiempo de entrenamiento de la cuarta red.

```
Época 399, loss entrenamiento 0.005386, loss validacion 0.144239
El loss de validacion empezo a aumentar en la epoca 399
Por lo tanto nos quedaremos con la red de la epoca 349

Entrenamiento finalizado
El entrenamiento tardó: 18.3329918384552

```

Al graficar loss de entrenamiento y validación por época se obtuvo el gráfico de la figura 10.

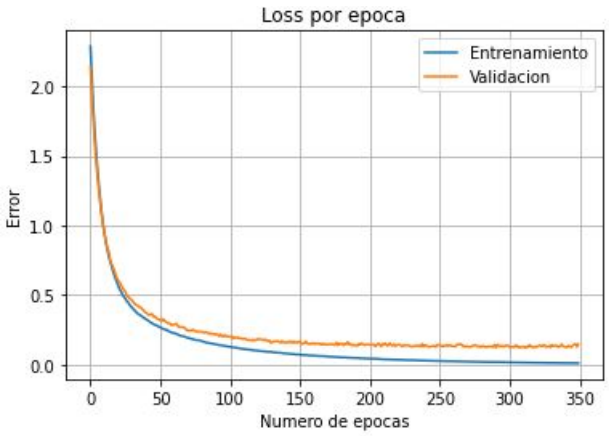

Figura 10: Loss por época para la cuarta red.

Al aplicar la red sobre el conjunto de entrenamiento se obtuvo el accuracy apreciado en el código 17 y la matriz de confusión correspondiente es la que se aprecia en la figura 11.

Código 17: Accuracy de la cuarta red sobre el conjunto de entrenamiento.

```
El accuracy de la red al usar el conjunto de entrenamiento es: 1.0

```

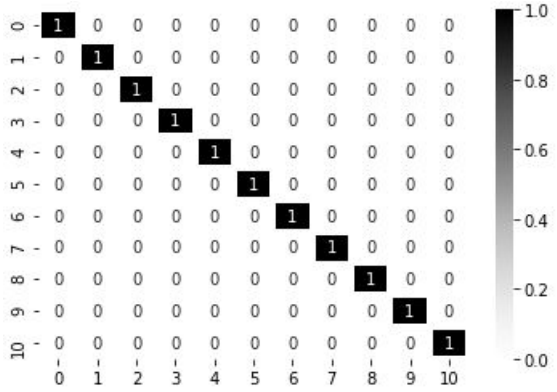

Figura 11: Matriz de confusión de la cuarta red sobre el conjunto de entrenamiento.

Finalmente el accuracy y la matriz de confusión de la red sobre el conjunto de validación son los que se observan en el código 18 y la figura 12.

Código 18: Accuracy de la cuarta red sobre el conjunto de validación.

```
El accuracy de la red al usar el conjunto de validacion es: 0.9509632224168126

```

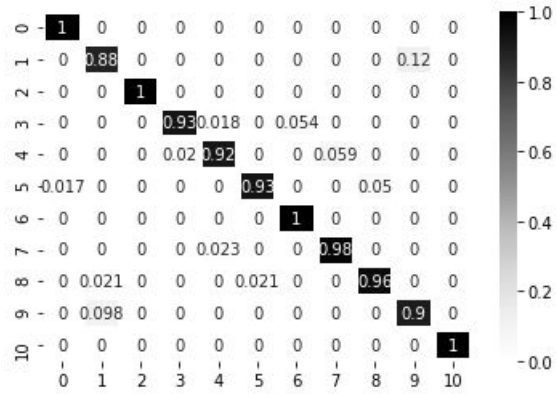

Figura 12: Matriz de confusión de la cuarta red sobre el conjunto de validación.

- **2 capas ocultas, ambas con 10 neuronas, utilizando función de activación ReLU y 1000 épocas:** La quinta red neuronal implementada corresponde a la que se puede observar en el código 19.

Código 19: Quinta red neuronal.

```
model = nn.Sequential(
  nn.Linear(48, 10),
  nn.ReLU(),
  nn.Linear(10,10),
  nn.ReLU(),
  nn.Linear(10,11)
```

```
)
```

Para la red neuronal observada en el código anterior se obtuvo el tiempo de entrenamiento observado en el código 20.

Código 20: Tiempo de entrenamiento de la quinta red.

```
Época 249, loss entrenamiento 0.069259, loss validacion 0.139805
El loss de validacion empezo a aumentar en la epoca 249
Por lo tanto nos quedaremos con la red de la epoca 199

Entrenamiento finalizado
El entrenamiento tardó: 14.1630220413208
```

El gráfico del loss de entrenamiento y de validación por época corresponde al observado en la figura 13.

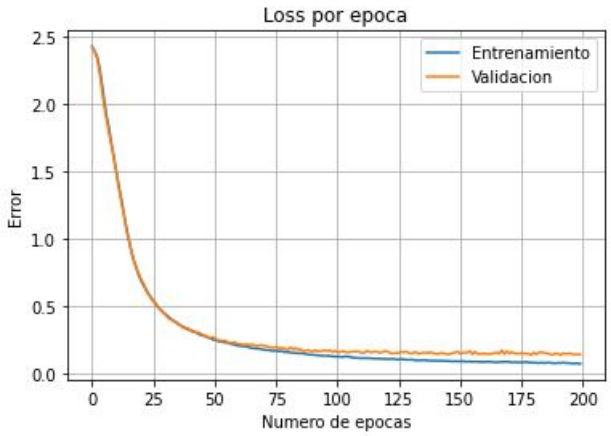

Figura 13: Loss por época para la quinta red.

El accuracy de la red aplicada sobre el conjunto de entrenamiento es el apreciado en el código 21, mientras que la matriz de confusión corresponde a la que se aprecia en la figura 14.

Código 21: Accuracy de la quinta red sobre el conjunto de entrenamiento.

```
El accuracy de la red al usar el conjunto de entrenamiento es: 0.9777227722772277
```

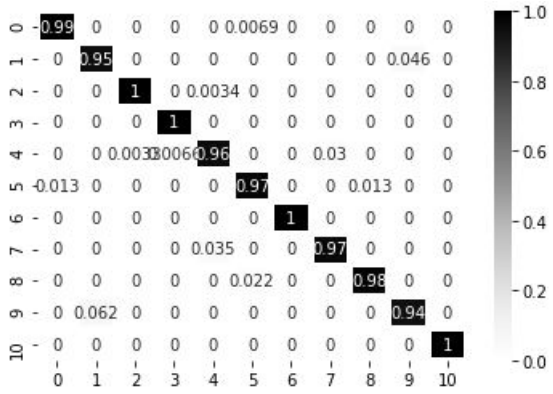

Figura 14: Matriz de confusión de la quinta red sobre el conjunto de entrenamiento.

Finalmente el accuracy y la matriz de confusión de la red al aplicarla sobre el conjunto de validación corresponden a los que se observan en el código 22 y la figura 15.

Código 22: Accuracy de la quinta red sobre el conjunto de validación.

```
El accuracy de la red al usar el conjunto de validacion es: 0.9422066549912435

```

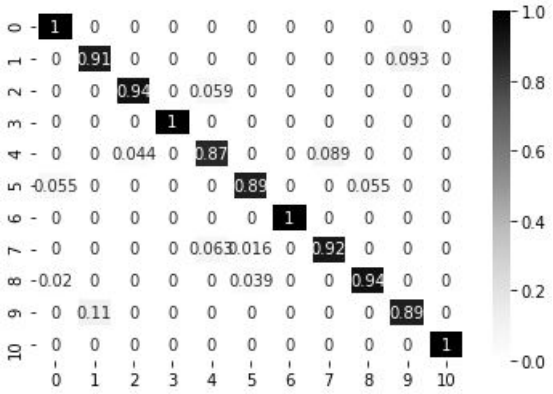

Figura 15: Matriz de confusión de la quinta red sobre el conjunto de validación.

- **2 capas ocultas, ambas con 40 neuronas, utilizando función de activación ReLU y 1000 épocas:** La última red neuronal implementada corresponde a la que se puede observar en el código 23.

Código 23: Sexta red neuronal.

```
model = nn.Sequential(
  nn.Linear(48, 40),
  nn.ReLU(),
  nn.Linear(40,40),
  nn.ReLU(),
  nn.Linear(40,11)
```

```
)
```

Para la red neuronal observada en el código anterior se obtuvo el tiempo de entrenamiento observado en el código 24.

Código 24: Tiempo de entrenamiento de la sexta red.

```
Época 149, loss entrenamiento 0.006075, loss validacion 0.164912
El loss de validacion empezo a aumentar en la epoca 149
Por lo tanto nos quedaremos con la red de la epoca 99

Entrenamiento finalizado
El entrenamiento tardó: 7.984354496002197
```

El gráfico del loss de entrenamiento y de validación por época corresponde al observado en la figura 16.

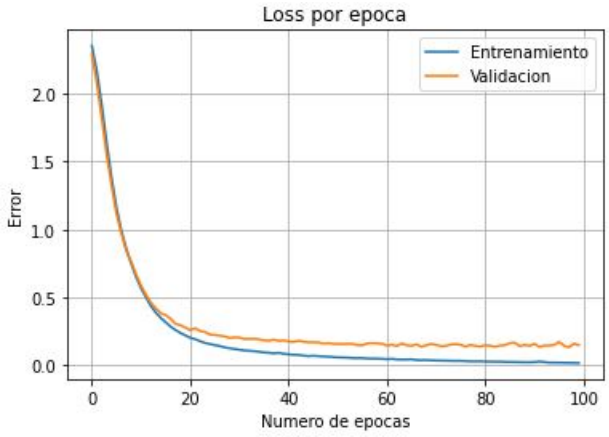

Figura 16: Loss por época para la sexta red.

El accuracy de la red aplicada sobre el conjunto de entrenamiento es el apreciado en el código 25, mientras que la matriz de confusión corresponde a la que se aprecia en la figura 17.

Código 25: Accuracy de la sexta red sobre el conjunto de entrenamiento.

```
El accuracy de la red al usar el conjunto de entrenamiento es: 0.9981435643564357
```

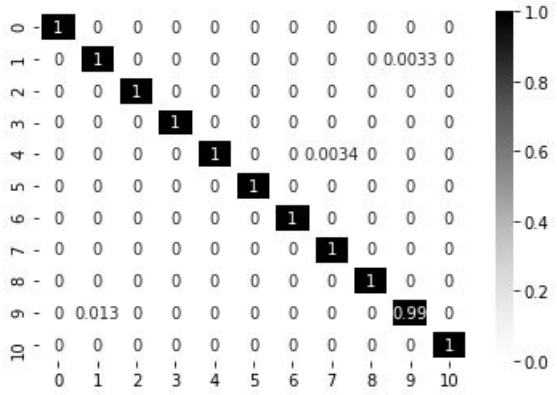

Figura 17: Matriz de confusión de la sexta red sobre el conjunto de entrenamiento.

Finalmente el accuracy y la matriz de confusión de la red al aplicarla sobre el conjunto de validación corresponden a los que se observan en el código 26 y la figura 18.

Código 26: Accuracy de la sexta red sobre el conjunto de validación.

```
El accuracy de la red al usar el conjunto de validacion es: 0.9509632224168126
```

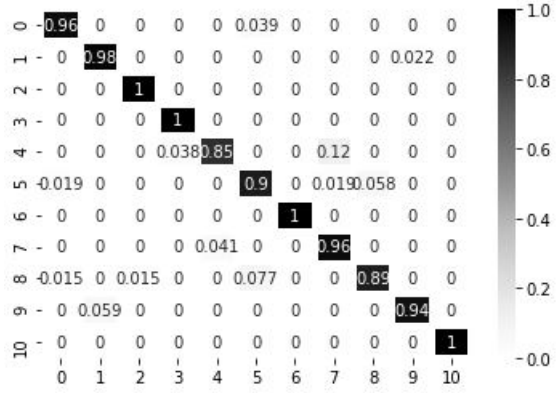

Figura 18: Matriz de confusión de la sexta red sobre el conjunto de validación.

2.3. Actividad 3

En esta actividad se expondrá la matriz de confusión y el accuracy de la red que obtuvo el mayor accuracy utilizando el conjunto de validación. Observando los accuracy en validación obtenidos durante la actividad anterior se logra apreciar que la red con 40 neuronas en capa oculta, utilizando una función Tanh y 1000 épocas, y la red con 2 capas ocultas de 40 neuronas cada una, utilizando una función de activación ReLU y 1000 épocas tienen exactamente el mismo accuracy en validación, por lo cual durante esta actividad se expondrán los accuracy y matriz de confusión de ambas redes sobre el conjunto de prueba. Es importante destacar que los resultados obtenidos durante la actividad anterior están sujetos a variaciones, con esto se apunta a que entre una ejecución del código y otra

los resultados que se obtengan podrían variar, y que por tanto se podría dar el caso de que otra red resulte con mayor accuracy en validación.

- **40 neuronas en capa oculta, utilizando función de activación Tanh y 1000 épocas:** Para esta red se obtuvo el accuracy que se puede observar en el código 27 y la matriz de confusión de la figura 19.

Código 27: Accuracy de la cuarta red sobre el conjunto de prueba.

```
El accuracy de la red al usar el conjunto de prueba es: 0.9587507365939893
```

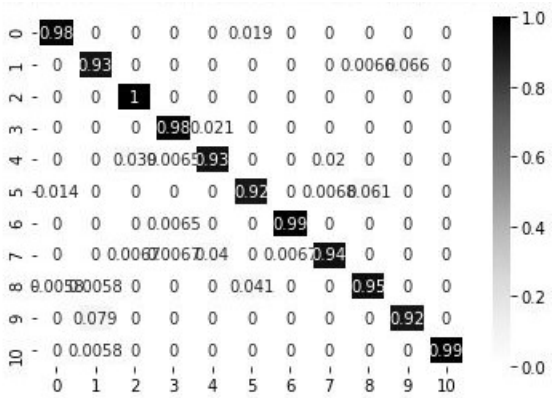

Figura 19: Matriz de confusión de la cuarta red sobre el conjunto de prueba.

- **2 capas ocultas, ambas con 40 neuronas, utilizando función de activación ReLU y 1000 épocas:** Para la sexta red evaluada en el conjunto de prueba se obtuvo el accuracy apreciado en el código 28 y la matriz de confusión de la figura 20.

Código 28: Accuracy de la sexta red sobre el conjunto de prueba.

```
El accuracy de la red al usar el conjunto de prueba es: 0.9616971125515615
```

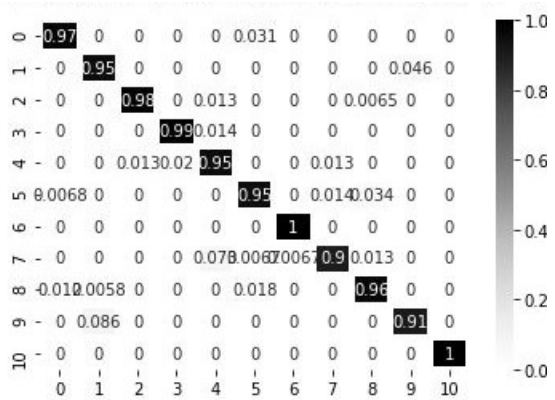

Figura 20: Matriz de confusión de la sexta red sobre el conjunto de prueba.

2.4. Actividad 4

A continuación se expondrá el análisis solicitado por el cuerpo docente, análisis que debe basarse en los resultados obtenidos durante las actividades anteriores.

- **a)** Para explicar como afecta la variación de neuronas en la capa oculta en el desempeño de la red se irá analizando de a pares, es decir, al usar una capa oculta con 10 y 40 neuronas y función de activación ReLU, luego tener una capa oculta con 10 o 40 neuronas y función de activación Tanh, y finalmente tener 2 capas ocultas, ambas con 10 neuronas o 40 y función de activación ReLU.

  Observando el tiempo de entrenamiento de las dos primeras redes, tiempo que se puede observar en el código 4 y 8, se logra observar que el hecho de tener más neuronas en capa oculta afecta de manera positiva, es decir, al tener más neuronas se obtiene un tiempo de entrenamiento menor, esto tiene sentido si se observa que el entrenamiento para la primera red se detuvo en la época 399, mientras que para la segunda red terminó en la 199. Luego si se observa el accuracy en el conjunto de entrenamiento de ambas redes (ver códigos 5 y 9) se desprende que tener más neuronas afecta de manera positiva al accuracy del conjunto de entrenamiento y a su vez, si se observan los códigos 6 y 10, se desprende exactamente el mismo análisis. Es decir, al usar una capa oculta y una función de activación ReLU se obtiene un mayor accuracy mientras mayor sea el número de neuronas en capa oculta.

  Observando la tercer y cuarta red se logra apreciar que este par de redes son más lentas, sobretodo cuando se tienen pocas neuronas en capa oculta, de esto se logra desprender que al tener 1 capa oculta con más neuronas se logra obtener un entrenamiento mucho más rápido. Posteriormente si se analizan accuracies de entrenamiento y validación se logra observar que la red con más neuronas en capa oculta presenta un mejor accuracy, esto se puede observar de manera directa al observar el accuracy en entrenamiento de la cuarta red, donde se obtuvo un accuracy de 1. Del análisis propuesto para una red con función de activación Tanh con distinta cantidad de neuronas en capa oculta se logra desprender que a mayor cantidad de neuronas existan en la capa oculta se logra un mejor accuracy tanto en validación como en entrenamiento, además de un entrenamiento más rápido.

  Finalmente si se observa el último par de redes, se logra observar un tiempo de entrenamiento menor al tener más neuronas en las capas ocultas (ver códigos 20 y 24) y al observar los accuracies se desprende que a más cantidad de neuronas se obtienen accuracies mejores.

  De todo lo anterior se logra obtener que a mayor cantidad de neuronas mejores son los tiempos de entrenamiento y los accuracies tanto en el conjunto de entrenamiento y validación, esto tendría sentido si se considera que una neurona es una "unidad de procesamiento", por lo cual al tener más neuronas se esperaría tener una mejor respuesta.

- **b)** Al observar las redes con una capa oculta y las redes con dos capas ocultas no se logra apreciar ninguna tendencia global, pues si se observan los tiempos de entrenamiento las redes con 2 capas ocultas presentan entrenamientos de 7 y 14 segundos, mientras que las de 1 capa presentan entrenamientos de 18, 11, 42 y 18 segundos, es decir, tener más capas no afecta

de manera global al tiempo de entrenamiento, si bien, si se reduce un poco el tiempo hay casos donde redes con 1 sola capa oculta se demoran menos en entrenar. Luego, si se observan accuracies basta con observar la red 4 con la 6, donde la primera de ellas tiene solo 1 capa oculta y obtuvo exactamente el mismo accuracy en el conjunto de validación que la red 6 que tenía 2 capas ocultas. Con todo lo anterior se puede decir que el tener más capas ocultas no afecta a las redes implementadas, sin embargo esto puede deberse a que la variación en cantidad de capas fue muy poca, si se hubiesen implementado redes con 1 capa oculta y otras con 10 capas ocultas es probable que si se hubiese detectado una tendencia en las redes de más capas.

- **c)** El tener distintas funciones de activación (ReLU y Tanh) no afecta de manera directa a los accuracies obtenidos, sino que afectan más al tiempo de entrenamiento, esto basta con observar la tercera red, donde el tiempo de entrenamiento obtenido es de 42 segundos y si utilizó una función de activación Tanh. Cabe destacar que los accuracies obtenidos para las redes con distintos tipos de función de activación no se ven altamente afectados debido a la cantidad de capas ocultas, pues es sabido que si se tuviesen muchas capas ocultas y se utilizara una función de activación Tanh el entrenamiento no podría realizarse debido al desvanecimiento del gradiente, sin embargo en esta tarea solo se implementaron 1 o 2 capas ocultas, motivo por el cual esto no está afectando a las redes implementadas.

- **d)** Al analizar tiempos de entrenamiento, accuracies y matrices de confusión de las distintas arquitecturas se logra destacar que las redes con funciones de activación ReLU tienden a ser más rápidas para entrenarse que las que utilizan una función de activación Tanh, sobretodo si tienen más neuronas en capas ocultas. Luego al observar las redes con 1 capa oculta y función de activación Tanh se logra apreciar que estas presentan un mayor accuracy que las redes con 1 capa oculta y función de activación ReLU, sin embargo, si se tienen redes con 2 capas ocultas y función de activación ReLU se obtienen accuracies comparables a las redes de 1 capa y función de activación Tanh. Finalmente el análisis desarrollado para los accuracies puede utilizarse para explicar las matrices de confusión, pues si se tiene un mejor accuracy la matriz de confusión también será mejor.

- **e)** Al observar accuracies y matrices de confusión de la cuarta red sobre el conjunto de validación y prueba se logra apreciar que el accuracy es mayor sobre el conjunto de prueba y por tanto la matriz de confusión de esta red en el conjunto de prueba también será mejor que la obtenida para el conjunto de validación. Luego, si se observa la sexta red nuevamente se tiene la tendencia de la red antes mencionada, es decir, el accuracy y la matriz de confusión obtenida es mejor al utilizar la red en el conjunto de prueba.

Nuevamente es importante destacar el hecho de que las explicaciones y análisis desarrollado durante esta actividad están sujetos a variaciones, pues si el código observado en el anexo (código 29) se ejecutase en diversas ocasiones es de esperar que los resultados obtenidos puedan variar.

3. Conclusión

A modo de conclusión se puede afirmar el cumplimiento de los objetivos propuestos en la tarea 4 del curso EL4106 Inteligencia Computacional, donde se lograron implementar diversos tipos de redes neuronales, así como también se logro evidenciar como es que los resultados de estas varían entre si.

Dentro de los aprendizajes obtenidos se destaca el uso de redes neuronales, pues es la primera vez que el estudiante se vio en el deber de implementar una. Además se destaca el uso de la librería *Torch*, librería que permite una implementación mucho más simple de una red neuronal, sin la necesidad de que el programador deba encargarse de programar algoritmos de alta complejidad como lo es el algoritmo de *Backpropagation*.

En cuanto a los resultados obtenidos se debe hacer mención de que las redes neuronales implementadas no siempre arrojarán los mismos resultados, por lo cual es probable que entre diversas ejecuciones del código la red que tenga mejores resultados varíe. También se destaca el hecho de que el uso de una mayor cantidad de neuronas permite obtener unos mejores resultados.

Finalmente, a modo de mejora, se podría intentar implementar una forma de armar los dataloaders para evitar que los resultados de las redes varíen entre las distintas ejecuciones del código, así como también podría buscarse una forma computacionalmente menos costosa de detener el entrenamiento, con esto se apunta a idear una forma en que el entrenamiento recurra a menos variables en memoria para obtener el mismo resultado.

4. Anexo

Código 29: Código implementado.

```
import pandas as pd
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

import numpy as np

from torch.utils.data import random_split

import sys

import sklearn.metrics

import seaborn as sns

import time

#---------------------------------------

from google.colab import files
uploaded = files.upload()  # sensorless_tarea4_train.txt

from google.colab import files
uploaded = files.upload()  # sensorless_tarea4_test.txt

#---------------------------------------

!ls

column_names = ["feat" + str(i) for i in range(48)]
column_names.append("class")

df_train_val = pd.read_csv('sensorless_tarea4_train.txt', names = column_names)
df_train_val["class"] = df_train_val["class"] - 1
df_train_val

df_test = pd.read_csv('sensorless_tarea4_test.txt', names = column_names)
df_test["class"] = df_test["class"] - 1
df_test

#---------------------------------------

model = nn.Sequential(
        nn.Linear(48, 40),
        nn.ReLU(),
        nn.Linear(40,40),
```

```
        nn.ReLU(),
        nn.Linear(40,11)
      )

device = torch.device('cuda')

model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

#----------------------------------------

# Crear datasets

feats_train_val = df_train_val.to_numpy()[:,0:48].astype(np.float32)
labels_train_val = df_train_val.to_numpy()[:,48].astype(int)
dataset_train_val = [ {"features":feats_train_val[i,:], "labels":labels_train_val[i]} for i in range(
    ↪ feats_train_val.shape[0]) ]

feats_test = df_test.to_numpy()[:,0:48].astype(np.float32)
labels_test = df_test.to_numpy()[:,48].astype(int)
dataset_test = [ {"features":feats_test[i,:], "labels":labels_test[i]} for i in range(feats_test.shape[0]) ]

n_train = int(df_train_val.shape[0]*0.85)
n_val = df_train_val.shape[0] - n_train

dataset_train, dataset_val = random_split(dataset_train_val, [n_train, n_val])

# Normalizar datos

fdata = []
i = 0

for x in dataset_train:
  fdata.append(x['features'])

fdata = np.array(fdata)

fmean= np.mean(fdata, axis=0)
fstd = np.std(fdata, axis=0)

for x in dataset_train:
  x['features'] = (x['features']-fmean) / fstd

for x in dataset_val:
  x['features'] = (x['features']-fmean) / fstd

for x in dataset_test:
  x['features'] = (x['features']-fmean) / fstd

```

```
97  # Crear dataloaders
98  dataloader_train = torch.utils.data.DataLoader(dataset_train, batch_size=128, shuffle=True,
        ↪ num_workers=0)
99  dataloader_val = torch.utils.data.DataLoader(dataset_val, batch_size=128, shuffle=True,
        ↪ num_workers=0)
100 dataloader_test = torch.utils.data.DataLoader(dataset_test, batch_size=128, shuffle=True,
        ↪ num_workers=0)
101
102 #----------------------------------------
103
104 tiempo_ini = time.time()
105
106 prom_loss_train = [] #Promedio de loss del entrenamiento para cada epoca
107 prom_loss_vali = [] #Promedio del loss de validacion para cada epoca
108 epocas = range(49,1000,50)
109 prom_loss_train_def = [] #Promedio de loss cada 50 epocas (Entrenamiento)
110 prom_loss_vali_def = [] #Promedio loss cada 50 epocas (Validacion)
111 for epoch in range(1000):
112     try:
113         model.train()
114
115         # Train on the current epoch
116         loss_epoch_train = [] #Acá se guarda el loss de cada dato de la epoca
117         for i, data in enumerate(dataloader_train, 0):
118             inputs = data["features"].to(device)
119             labels = data["labels"].to(device)
120
121             # zero the parameter gradients
122             optimizer.zero_grad()
123
124             # forward + backward + optimize
125             outputs = model(inputs)
126
127             loss_train = criterion(outputs, labels) #calculo del loss de entrenamiento por dato
128             loss_epoch_train.append(loss_train)     #guardamos el loss del dato en un vector
129             loss_train.backward()
130             optimizer.step()
131         prom_train = sum(loss_epoch_train)/len(loss_epoch_train) #Calculamos el loss promedio de
            ↪ la epoca
132         prom_loss_train.append(prom_train) #Guardamos el promedio en un vector
133
134         # Compute validation loss and accuracy for current epoch
135         model.eval()
136
137         with torch.no_grad():
138             loss_epoch_vali = [] #Acá se guarda el loss de cada dato de la epoca
139             for i, data in enumerate(dataloader_val, 0):
140                 inputs = data["features"].to(device)
141                 labels = data["labels"].to(device)
142
143                 outputs = model(inputs)
```

```
144             # Calcular loss de validación
145             loss_vali = criterion(outputs, labels) #calculo del loss de validación por dato
146             loss_epoch_vali.append(loss_vali)      #guargamos el loss del dato en un vector
147         prom_vali = sum(loss_epoch_vali)/len(loss_epoch_vali) #Calculamos el loss promedio de la
        ↪ epoca
148         prom_loss_vali.append(prom_vali)
149
150         # Imprimir: numero de época, loss de entrenamiento y loss de validación
151         # Se debe usar sys.stdout.write() para que la línea de texto se sobreescriba en vez de imprimirse
        ↪  línea por línea
152         # No se debe guardar checkpoints en cada época (guardarlos cada 50 épocas)
153
154         sys.stdout.write("\rÉpoca %d, loss entrenamiento %f, loss validacion %f" % (epoch,
        ↪ prom_train, prom_vali))
155
156         #Hacemos el checkpoint cada 50 epocas y revisamos si debemos interrumpir el entrenamiento
157         if epoch in epocas:
158           torch.save(model, 'red'+str(epocas.index(epoch))+'.pt')
159           s_train = 0 #Suma de los loss de las 50 epocas anteriores (entrenamiento)
160           s_vali = 0 #Suma de los loss de las 50 epocas anteriores (validacion)
161           for i in range(epoch-50,epoch):
162             s_train = prom_loss_train[i]+s_train
163             s_vali = prom_loss_vali[i]+s_vali
164           prome_train = s_train/50 #Loss promedio de las 50 epocas (Entrenamiento)
165           prome_vali = s_vali/50 #Loss promedio de las 50 epocas (Validacion)
166           prom_loss_train_def.append(prome_train)
167           prom_loss_vali_def.append(prome_vali)
168           #Si el loss de validacion actual es mayor que el de la vez pasada termino
169           #El entrenamiento
170           if prom_loss_vali_def[len(prom_loss_vali_def)-1] > prom_loss_vali_def[len(
        ↪ prom_loss_vali_def)-2]:
171             print('\nEl loss de validacion empezo a aumentar en la epoca '+str(epoch))
172             print('Por lo tanto nos quedaremos con la red de la epoca '+str(epoch-50))
173             model = torch.load('red'+str(epocas.index(epoch)-1)+'.pt') #Nos quedamos con la red
        ↪ anterior
174             break
175
176       except KeyboardInterrupt:
177         print("\nEntrenamiento interrumpido")
178         break
179
180 print('\nEntrenamiento finalizado')
181 tiempo_fin = time.time()
182 print("El entrenamiento tardó: "+str(tiempo_fin-tiempo_ini))
183
184 #Grafico del loss por epoca
185 final = len(prom_loss_train)-50 #Esto es para graficar hasta el momento en que el loss fue minimo
186 plt.plot(range(final), prom_loss_train[:final], label='Entrenamiento')
187 plt.plot(range(final), prom_loss_vali[:final], label='Validacion')
188 plt.grid()
189 plt.legend(loc='upper right')
```

```
plt.xlabel('Numero de epocas')
plt.ylabel('Error')
plt.title('Loss por epoca')

#---------------------------------------

predict_train = []
class_train = []
for i, data in enumerate(dataloader_train, 0):
  inputs = data["features"].to(device)
  labels = data["labels"].to(device)

  #Obtenemos las salidas
  outputs = model(inputs)
  #Obtenemos las predicciones
  pred = outputs.cpu().argmax(axis=1)
  #A continuacion obtendremos la prediccion hecha para cada elemento y la
  #guardaremos en un vector, lo mismo hacemos con los labels
  for i in range(len(pred)):
    predict = pred[i].numpy().item()
    clas = labels[i].cpu().numpy().item()
    predict_train.append(predict)
    class_train.append(clas)

accuracy_train = sklearn.metrics.accuracy_score(class_train, predict_train, normalize=True)
print("El accuracy de la red al usar el conjunto de entrenamiento es: "+str(accuracy_train))
matrix_train = sklearn.metrics.confusion_matrix(class_train, predict_train, normalize='true')
sns.heatmap(matrix_train, linewidths=3, annot=True, cmap="Greys")

#---------------------------------------

predict_vali = []
class_vali = []
for i, data in enumerate(dataloader_val, 0):
  inputs = data["features"].to(device)
  labels = data["labels"].to(device)

  #Obtenemos las salidas
  outputs = model(inputs)
  #Obtenemos las predicciones
  pred = outputs.cpu().argmax(axis=1)
  #A continuacion obtendremos la prediccion hecha para cada elemento y la
  #guardaremos en un vector, lo mismo hacemos con los labels
  for i in range(len(pred)):
    predict = pred[i].numpy().item()
    clas = labels[i].cpu().numpy().item()
    predict_vali.append(predict)
    class_vali.append(clas)

accuracy_vali = sklearn.metrics.accuracy_score(class_vali, predict_vali, normalize=True)
print("El accuracy de la red al usar el conjunto de validacion es: "+str(accuracy_vali))
```

```
241 matrix_vali = sklearn.metrics.confusion_matrix(class_vali, predict_vali, normalize='true')
242 sns.heatmap(matrix_vali, linewidths=3, annot=True, cmap="Greys")
243
244 #---------------------------------------
245
246 predict_test = []
247 class_test = []
248 for i, data in enumerate(dataloader_test, 0):
249   inputs = data["features"].to(device)
250   labels = data["labels"].to(device)
251
252   #Obtenemos las salidas
253   outputs = model(inputs)
254   #Obtenemos las predicciones
255   pred = outputs.cpu().argmax(axis=1)
256   #A continuacion obtendremos la prediccion hecha para cada elemento y la
257   #guardaremos en un vector, lo mismo hacemos con los labels
258   for i in range(len(pred)):
259     predict = pred[i].numpy().item()
260     clas = labels[i].cpu().numpy().item()
261     predict_test.append(predict)
262     class_test.append(clas)
263
264 accuracy_test = sklearn.metrics.accuracy_score(class_test, predict_test, normalize=True)
265 print("El accuracy de la red al usar el conjunto de prueba es: "+str(accuracy_test))
266 matrix_test = sklearn.metrics.confusion_matrix(class_test, predict_test, normalize='true')
267 sns.heatmap(matrix_test, linewidths=3, annot=True, cmap="Greys")
```