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

Tarea 5

Clasificación de objetos usando CNNs.

Integrantes: Joaquín Zepeda
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
Santiago de Chile

Índice de Contenidos

Índice de Figuras

Índice de Tablas

1. Introducción

La detección de personas y de rostros no es algo simple en el ámbito del procesamiento de imágenes, existen múltiples técnicas que buscan reconocer caras y personas las cuales son utilizadas en seguridad, universidades, lugares públicos, etc. Las características extraídas de la imagen son importantes para el reconocimiento de estas, por ejemplo cara, brazos, torso, etc. El objetivo de esta tarea es diseñar y construir un sistema de clasificación de objetos usando redes neuronales convolucionales (CNNs) utilizando la librería Pytorch, esto con el fin de poner en práctica los conceptos vistos en clases, poner en práctica las habilidades de programación para resolver un problema real y poder apreciar en diferentes situaciones los resultados de estos métodos y de los distintos clasificadores. Para esto se utilizará el dataset CIFAR10 [1], el cual contiene 10 categorías/clases de objetos, cada uno de los cuales corresponde a una imagen RGB de 32x32 píxeles.
Esta tarea se desarrolla utilizando el lenguaje de programación Python. A continuación se describe el procedimiento de la tarea y se muestran y analizan los resultados en la sección de Desarrollo para luego finalizar con las conclusiones.

2. Desarrollo

CIFAR10 es un dataset el cual contiene 10 clases de objetos,cada uno de los cuales corresponde a una imagen RGB de 32x32 píxeles. Para está tarea se utilizarán 3 batches: 'data_batch_1' para entrenamiento, 'data_batch_2' para validación y 'test_batch' para Prueba, cada uno de estos conjuntos tiene 10000 imágenes.

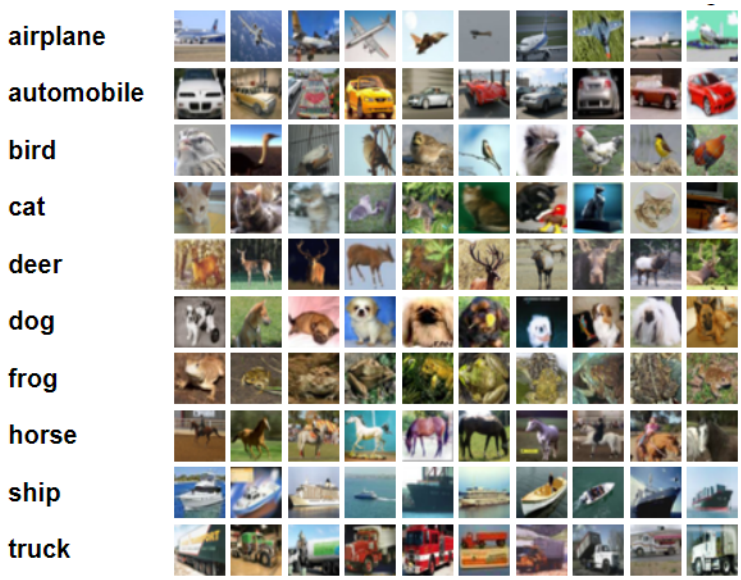


Figura 1: 10 ejemplos por clase.

En los items 2., 3. y 4.. se implementan distintas configuraciones de redes neuronales, en donde en el item 2. se realiza una configuración con 3 capas convolucionales, el item 3. con la misma configuración del item 2. pero variando el número de neuronas en la última capa oculta y en el item 4 se prueban 2 configuraciones agregando capas convolucionales. Es importante destacar que las configuraciones de las redes tienen una capa fully-connected intermedia que también depende del valor de N, esto con el fin de adaptar de mejor manera la red para cada N.

Tabla 1: Configuraciones de las redes neuronales en los items 2., 3. y 4.. convX-Y se refiere a una capa convolucional con Y filtros de (X,X).FC se refiere a Fully-Connected.

Configuraciones redes neuronales		
Red inicial (3 capas convolucionales)	Red con 4 capas convolucionales	Red con 5 capas convolucionales
Input (3x32x32)		
conv3-64+ReLU+BN	conv3-64+ReLU+BN	conv3-64+ReLU+BN
conv3-64+ReLU+BN	conv3-64+ReLU+BN	conv3-64+ReLU+BN
Maxpool 2x2		
conv3-128+ReLU+BN	conv3-128+ReLU+BN	conv3-128+ReLU+BN
Maxpool 2x2		
	conv3-256+ReLU+BN	conv3-256+ReLU+BN
	Maxpool 2x2	conv3-512+ReLU+BN
		Maxpool 2x2
FC-1024		
FC-24xN		
FC-N		
Softmax		

A continuación se describe cada item del enunciado, describiendo códigos/algoritmos importantes, los resultados y los respectivos análisis de resultados.

1. Implementar el código para que pytorch acceda a los datasets, para el conjunto de entrenamiento, validación y prueba.

Los datos se descargan desde la página de CIFAR10 https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz, luego se deben descomprimir para tener todos los batches en archivos binarios, por lo que se debe codificar, para lo cual se utiliza la siguiente función:

```
def unpickle(file):
    # función que decodifica un archivos y retorna un diccinoario # con sus datos
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='latin1')
    return dict
```

Está función decodifica los archivos y retorna un diccionario, de acá se puede obtener los datos, correspondientes a arreglos de 3072 (1024 valores por cada canal) y los labels, correspondientes a la clase de cada dato. Para visualizar una imagen de los conjuntos cargados el arreglo de 3072 se debe realizar un reshape (cambio de forma) para que quede de 32x32x3.
Se desarrollan 3 clases para cada conjunto (entrenamiento, validación y prueba) las cuales son análogas cambiando solo en el nombre del archivo que se lee (***), por lo que a continuación se describe solo la clase de CIFAR10Train.

```
from torch.utils.data import Dataset
```

```
class CIFAR10Train(Dataset):
    def __init__(self, path):
        # Constructor, debe leer el archivo data_batch_1 dentro de la carpeta indicada (este archivo
        ↪ se usará para el set de entrenamiento)
        self.dict_data = unpickle(path+'data_batch_1') #***
        self.labels,self.filenames = self.dict_data['labels'],self.dict_data['filenames']

    def __len__(self):
        # Debe retornar el número de imágenes en el dataset de entrenamiento
        return len(self.filenames)

    def __getitem__(self, index):
        data = self.dict_data['data']
        #retorna un par label, image dado un indice. Donde image #es un arreglo de 3x32x32
        #escalamineto lineal realizado a todos los pixeles
        datax = -1 +2/255*data[index]
        img = np.reshape(datax, (3,32,32))
        return self.labels[index], img
```

A partir de esto se generan los 3 dataset's para luego generar los 3 contenedores iterables dataloader's, se utiliza un $BATCH\_SIZE = 256$ en todos los conjuntos. A continuación se muestra lo más importante:

```
trainDataset = CIFAR10Train("/content/cifar-10-batches-py/")
valDataset = CIFAR10Val("/content/cifar-10-batches-py/")
testDataset = CIFAR10Test("/content/cifar-10-batches-py/")

BATCH_SIZE = 256
train_loader = torch.utils.data.DataLoader(trainDataset, batch_size=BATCH_SIZE, shuffle=True,
    ↪ num_workers=4, pin_memory=True)
val_loader = torch.utils.data.DataLoader(valDataset, batch_size=BATCH_SIZE, shuffle=True,
    ↪ num_workers=4, pin_memory=True)
test_loader= torch.utils.data.DataLoader(testDataset, batch_size=BATCH_SIZE, shuffle=True,
    ↪ num_workers=4, pin_memory=True)
```

2. Implementar una red inicial.

- 2.a Se genera una función train (parte de esta función se basaron en códigos provistos por el enunciado y del curso Deep learning que estoy cursando actualmente) la cual se encarga de entrenar la red neuronal. Las redes sufren sobreajuste si la cantidad de parámetros es grande, a medida que el entrenamiento va progresando. Para poder evitar el sobreajuste, se implementa un enfoque basado en patience. En el entrenamiento se van guardando checkpoints cada vez que el loss actual sea menor que el loss de validación existente, el cual se retorna luego del entrenamiento. Esta función también gráfica la el error en función de las épocas. La arquitectura de esta red se puede observar en la tabla 1.

```
    # Parte de los prints de los accuracy's y de los loss se basaron en modelos
    # del curso Deep learning que estoy cursando actualmente.
    def train(net, optimizer, num_epocas):
```

```
    inicio = time.time()
    #copiamos el modelo utilizando la libreria copy
    best_model_wts = copy.deepcopy(net.state_dict())
    train_losses = []
    train_counter = []
    train_accuracy = []
    val_losses = []
    val_accuracy = []
    best_acc = 0.0
    best_loss = 2e32
    for epoch in range(num_epocas):
      print('Epoch {}/{}'.format(epoch, num_epocas-1))
      print('-' * 10)

      net.train() #Modo entrenamiento

      running_loss = 0.0
      running_corrects = 0.0
      for i, data in enumerate(train_loader, 0): # Obtener batch
          labels = data[0].cuda()
          inputs = data[1].cuda().float()
          optimizer.zero_grad()
          outputs = net(inputs) #salidas de la red
          preds = outputs.argmax(axis=1) #predicciones
          loss = criterion(outputs, labels)
          loss.backward()
          optimizer.step()

          running_loss += loss.item() * inputs.size(0)
          running_corrects += torch.sum(preds == labels.data)

      epoch_loss = running_loss /len(train_loader.dataset) #promedio de error
      epoch_acc = running_corrects.double() / len(train_loader.dataset) #promedio de
    ↪ accuracy
      train_losses.append(epoch_loss)
      train_counter.append(epoch)
      train_accuracy.append(epoch_acc)

      print('Train Loss: {:.4f} Acc: {:.4f}'.format(epoch_loss, epoch_acc))

      #Validacion
      net.eval()

      running_loss = 0.0
      running_corrects = 0.0
      for labels,inputs in val_loader:
          inputs = inputs.to(device).float()
          labels = labels.to(device)
          with torch.set_grad_enabled(False):
              outputs = net(inputs)
              preds = outputs.argmax(axis=1)
```

```
54              val_loss = criterion(outputs, labels)
55              #val_losses.append(val_loss.item())
56              #correct += pred.eq(target.data.view_as(pred)).sum()
57
58              running_loss += loss.item() * inputs.size(0)
59              running_corrects += torch.sum(preds == labels.data)
60
61          epoch_loss = running_loss /len(val_loader.dataset)
62          epoch_acc = running_corrects.double() / len(val_loader.dataset)
63          val_losses.append(epoch_loss)
64          val_accuracy.append(epoch_acc)
65          print('Val Loss: {:.4f} Acc: {:.4f}'.format(epoch_loss, epoch_acc))
66
67          #chekpoint
68          if epoch_loss < best_loss:
69              best_loss = epoch_loss
70              best_model_wts = copy.deepcopy(net.state_dict())
71
72          # early stopping, si el error aumenta más de 5 veces respecto al menor error,
73          # terminamos el entrenamiento
74          if epoch_loss > best_loss*5:
75              print('\n'+'-' * 10+'Early Stopping'+'-' * 10+'\n')
76              break
77
78      print('Best val loss: {:.4f}'.format(best_loss))
79      plt.figure()
80      #2b. Graficar las curvas de loss de entrenamiento y validación
81      plt.title("Error en cada epoca")
82      plt.plot(train_counter, train_losses, label='Entrenamiento',color='blue')
83      plt.plot(train_counter,val_losses, label='Validacion',color='red')
84      plt.xlabel("Epochs")
85      plt.ylabel("Loss")
86      plt.legend()
87      plt.show()
88
89      final = time.time()
90      print('Training complete in {:.0f}m {:.0f}s'.format((final-inicio)//60, (final-inicio) % 60))
91
92      net.load_state_dict(best_model_wts)
93      return net
94
```

Está función se utiliza para los items siguientes también, en donde será importante el early stopping implementado (si el error en la época es 5 veces mayor al mejor error registrado, termina el entrenamiento).

- En la figura 2 se puede observar como el error en validación va tendiendo a cero, esta curva tiene ciertos aumentos de errores pero la tendencia sigue disminuyendo.

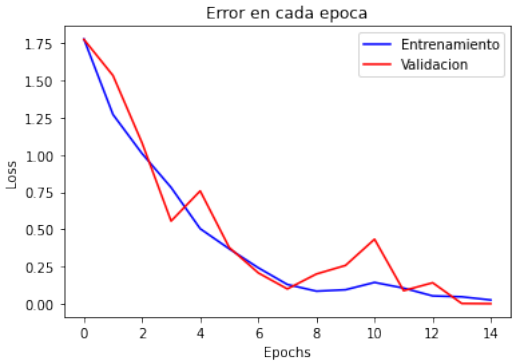
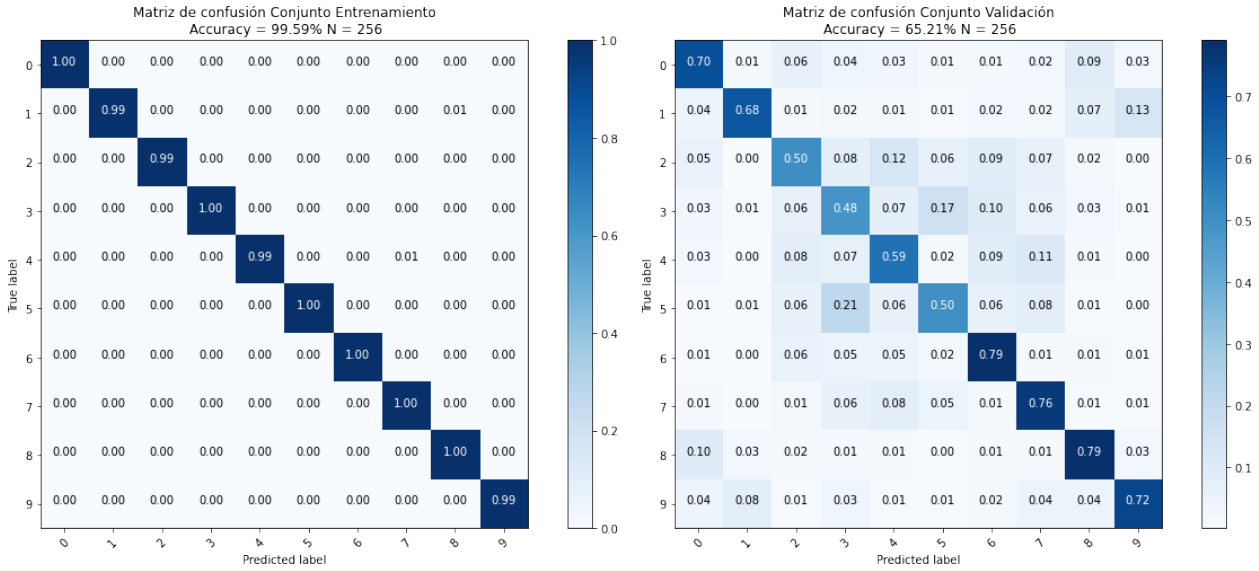


Figura 2: Curva Loss de Entrenamiento y Validación en cada época en la red inicial.

- En la figura 3, se pueden observar las matrices de confusión de entrenamiento y validación, en la matriz de confusión de entrenamiento se observa el **overfitting** que tiene esta red, llegando a tener un accuracy en entrenamiento de 99.59 %, es decir, la red se está aprendiendo de memoria los ejemplos de entrenamiento. A pesar de esto, los resultados en validación que se muestran en la matriz de confusión de validación tienen clasifica de buena manera ciertas clases, como la 6, 7 y 8 (Rana, Caballo y Barco), pero clasifica mal otras como la 2, 3 y 4 (Ave, Gato y Ciervo). A pesar de esto un 65.21 % de Accuracy no es un mal desempeño, pero puede mejorar.



(a) (b)

Figura 3: Matrices de confusión normalizadas de la red inicial, utilizando N=256.

3. Implementar una segunda red cambiando el valor de N.

Para determinar una red con un buen desempeño, se probó la configuración de la red inicial variando el valor de N, guardando el valor del Accuracy en validación y los modelos para así elegir el

N que maximiza este valor. En la figura 4 se puede observar que los mejores resultados se obtienen con N menores a 100, siendo el punto óptimo N=64 (se eligieron valores de N múltiplos de a 2). Se utilizaron 20 épocas para todos modelos y los distintos valores de N.

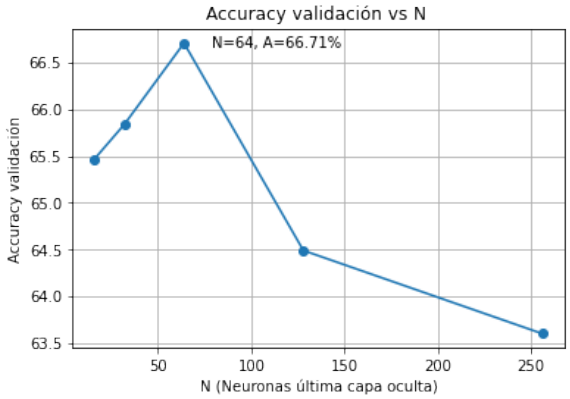


Figura 4: Accuracy versus Número de N neuronas en la ultima capa.

```
models = []
accuracys = []
n_list = [16,32,64,128,256] #valores de N a probar
for N in n_list:
    print(f'\nModel con N={N}')
    net = MyNet(N)
    net.cuda()
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
    best_net = train(net, optimizer, num_epocas=20) #20 epocas
    models.append(best_net) #guardamos el mejor checkpoint
    accuracys.append(evaluar_red(best_net,N,plot=False)) #guardamos el accuracy
```

En la figura 5 se puede observar que ocurrió el Early stopping, pues luego de la época 14, el error en validación aumentó mucho y se detuvo el entrenamiento.

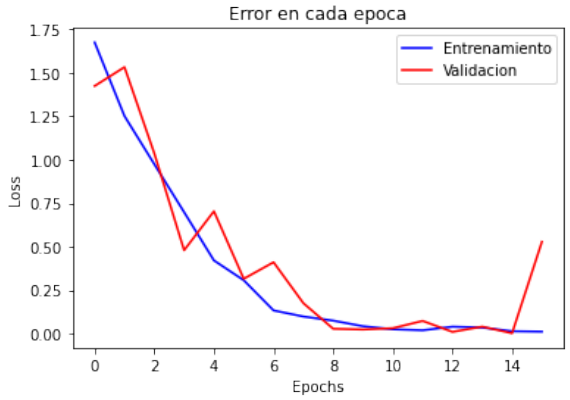
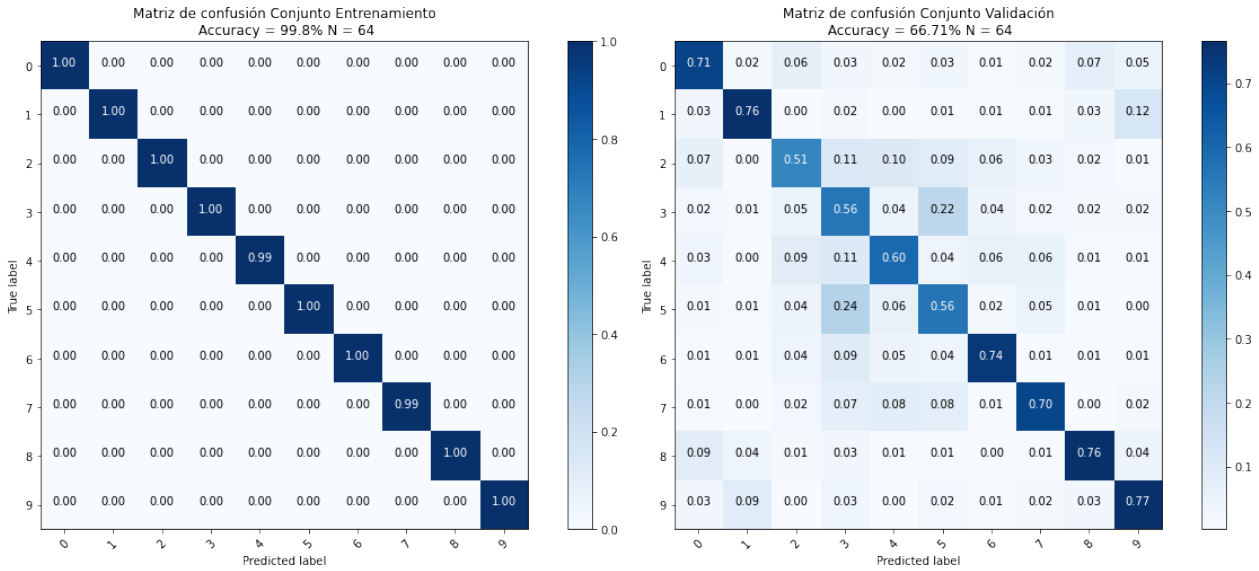


Figura 5: Curva Loss mejor N=64.

Como se puede observar en la figura 6, se lograron mejores resultados que con la red inicial al utilizar un N=64, en efecto, el accuracy aumentó en aproximadamente un 1.5 %, logrando asi que todas las clases tuvieran más del 50 % de clasificaciones correctas, pero algunas siguen estando cerca del 50 %, lo cual podría mejorar.

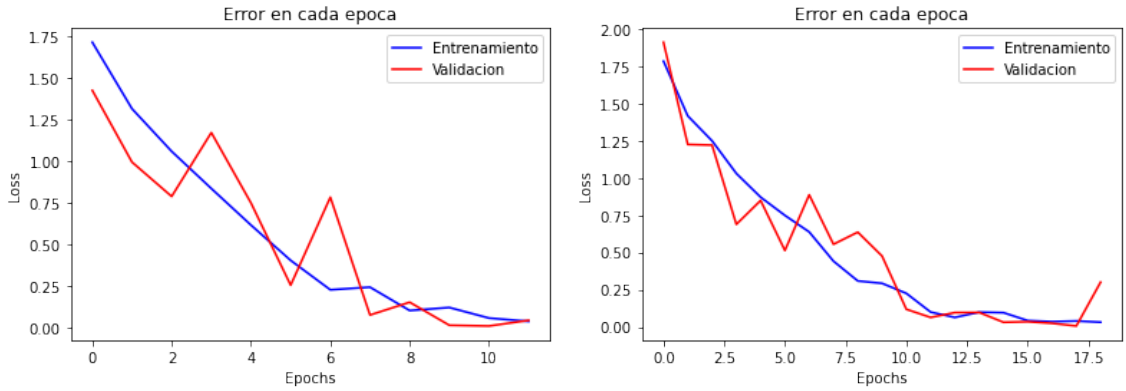


(a) (b)

Figura 6: Matrices de confusión normalizadas de la red con el mejor N, utilizando el mejor N.

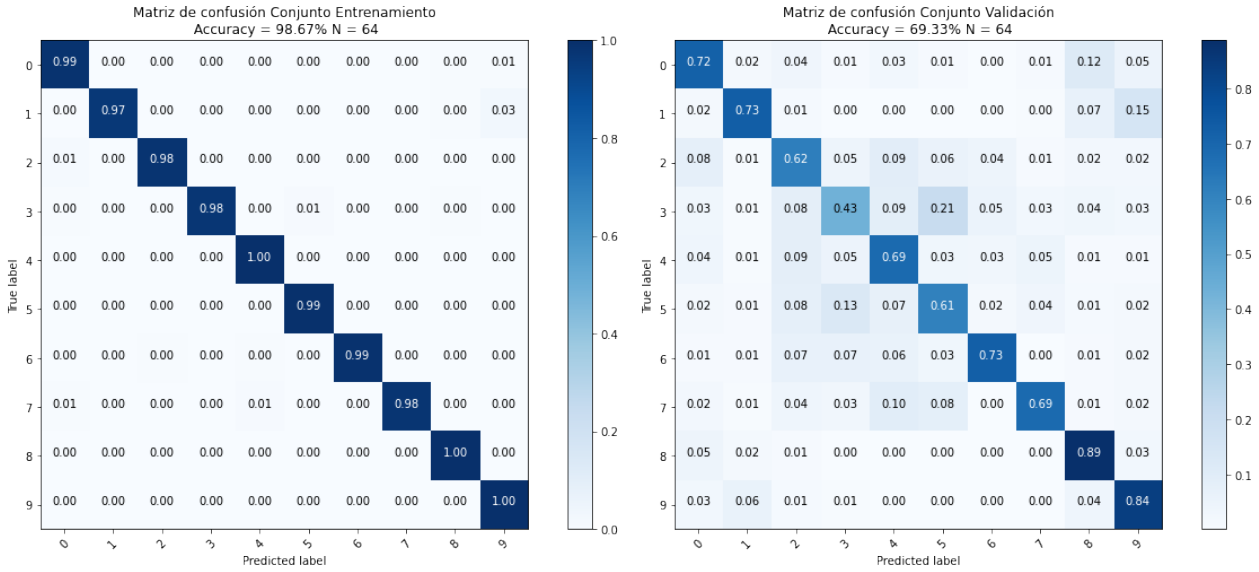
4. Implementar una segunda red cambiando el número de capas convolucionales.

Para está sección se utilizó el mejor valor de N, correspondiente a 64. Luego se implementaron 2 redes neuronales, una con 4 capas y otra con 5 capas convolucionales, lo cual corresponde a agregarle 1 y 2 capas a la red inicial respectivamente. La arquitectura de estas redes se puede observar en la tabla 1. Como se puede observar en las figuras 8 y 9, al aumentar el número de capas convolucionales aumentan las características extraídas y esto disminuye levemente el accuracy en el conjunto de entrenamiento (lo cual se buscaba para disminuir el overfitting), pero aumenta el accuracy en validación, llegando a un 69.33 % y 71.42 % respectivamente, mejorando considerablemente el desempeño en comparación con la red inicial.
En la figura 7 se puede observar que el error en validación de la red con 5 capas convolucinoales disminuye de forma más lenta en comparación a la red con 4 capas, a pesar de esto está tiene un número mayor de aumentos (o peaks) en validación, pero estos son de menor tamaño que en la red de 4 capas.



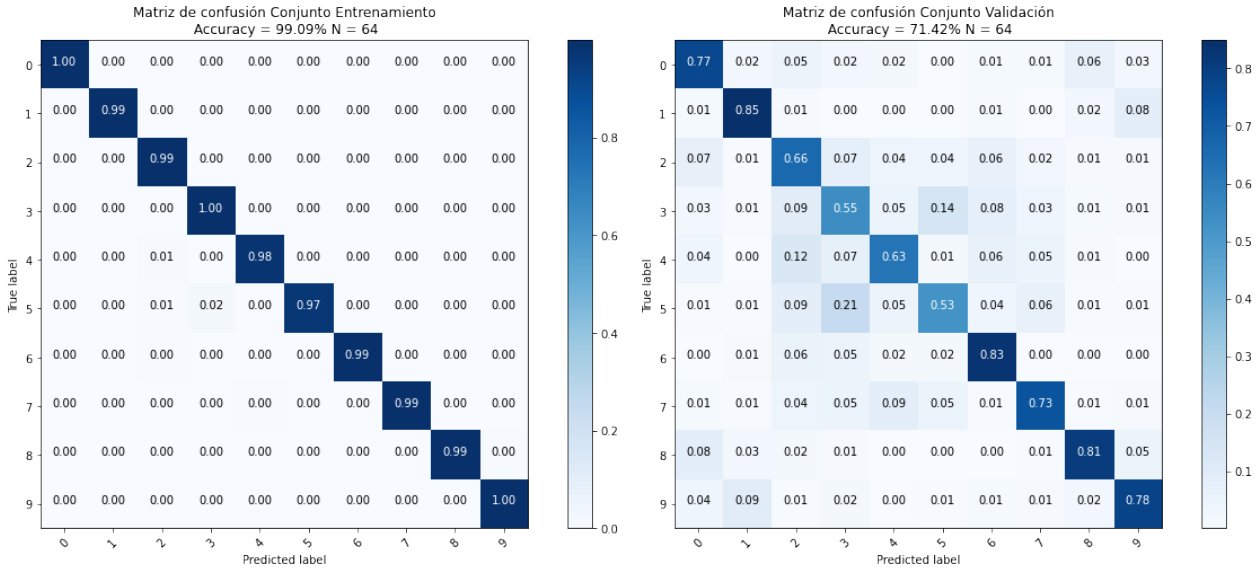
(a) 4 capas. (b) 5 capas.

Figura 7: Curvas de Loss, para las redes de con 4 y 5 capas convolucionales.



(a) (b)

Figura 8: Matrices de confusión normalizadas de la red con 4 capas, utilizando el mejor N.



(a) (b)

Figura 9: Matrices de confusión normalizadas de la red con 5 capas, utilizando el mejor N.

Luego de observar estás 4 matrices de confusión, las curvas de los y los resultados de accuracy, se puede concluir que la red que genera mayores resultados corresponde a la red con 5 capas convolucionales y con 64 neuronas.

5. Evaluar resultados en el conjunto de Prueba.

Con la mejor red encontrada, se evaluó la red sobre el conjunto de prueba. obteniendo un 71.01 % de Accuracy, logrando incluso clasificar 4 clases sobre el 80 % de clasificaciones correctas, correspondiendo a las clases Automóvil, Rana, Barco y Camión (1, 6, 8 y 9 respectivamente). A pesar de esto, la red sigue sufriendo de Overfitting.

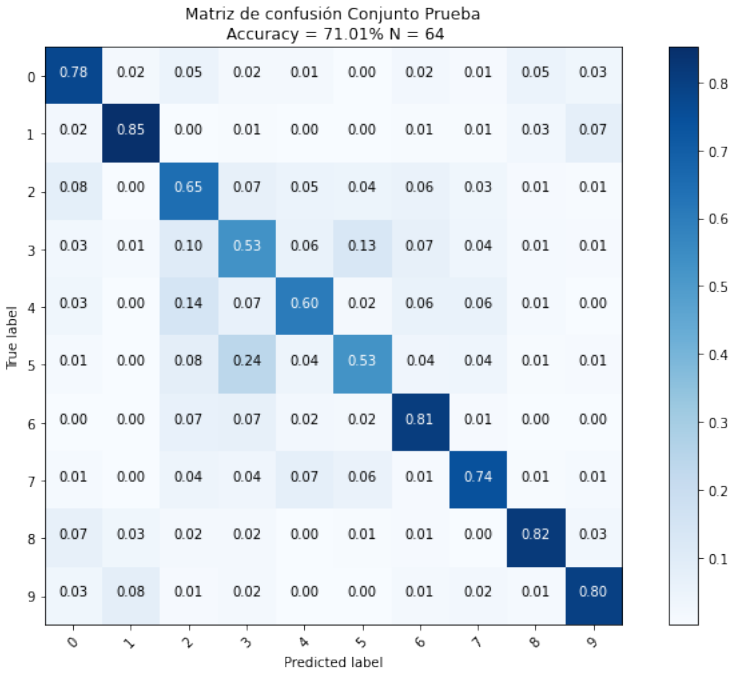


Figura 10: Matriz de confusión utilizando el mejor modelo encontrado en las secciones anteriores.

3. Conclusión

Luego del desarrollo de esta tarea, fue posible implementar un clasificador de objetos usando CNNs, cumpliendo el objetivo de esta, además los clasificadores desarrollados presentaron buenos resultados en el conjunto de prueba (sobre el 70 %), evidenciando así la efectividad de las redes neuronales convolucionales, a pesar de esto, estas redes siguen sufriendo Overfitting, lo cual se podría mejorar utilizando técnicas como Data Augmentation o capas de DropOut en las capas fully connected. Además fue posible comprender las redes CNN, sus ventajas y el como implementarlas usando la librería Pytorch. Estas herramientas son bastante útiles para nuestra formación como ingenieros pues se puede aplicar a un sin fin de proyectos.
Se recomienda el uso de la red CNN con 5 capas convolucionales y 64 neuronas en la capa oculta en caso de querer realizar un proyecto similar, pues fue el que presentó mejores resultados identificando objetos.
Se pusieron en practica los conceptos y técnicas vistas en clases, programarlas en Python, utilizando uno de las librerias más utilizadas para realizar redes neuronales tal como es Pytorch. La mayor dificultad de esta tarea fue el de hacer calzar las dimensiones de la red neuronal, el implementar el early stopping y el de ir guardando los checkpoint.

Referencias

[1] The CIFAR-10 dataset. Disponible en: https://www.cs.toronto.edu/~kriz/cifar.html

4. Anexos

```
# -*- coding: utf-8 -*-
"""Tarea5_imagenes.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1Gko2mgN7qFImaH2lCugIoRudM07G1Xph

# Desarrollo por Joaquin Zepeda V.

Tarea 5 EL7008 - Clasificación de objetos usando CNNs.

El objetivo de esta tarea es implementar un sistema de clasificación de objetos usando redes neuronales
convolucionales (CNNs). En esta tarea se debe usar la librería pytorch para poder generar los tensores que
corresponden a las imágenes y sus labels (etiquetas), además de implementar arquitecturas de red y c ódigos
para entrenamiento y evaluación
"""

!wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz

!tar -xzvf cifar-10-python.tar.gz

import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import numpy as np
import matplotlib.pyplot as plt
import pickle
import copy
from torchvision import datasets, models, transforms
import time
import os
import random

SEED = 1234


torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)
np.random.seed(SEED)
random.seed(SEED)

```

```
device = ('cuda' if torch.cuda.is_available() else 'cpu')

device

"""# Sección 1. Implementar el código para que pytorch acceda a los datasets, para el conjunto de entrenamiento, validación y prueba."""

def unpickle(file):
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='latin1')
    return dict

L = unpickle("/content/cifar-10-batches-py/data_batch_1")

data,labels,filenames = L['data'],L['labels'],L['filenames']
metadata = unpickle("/content/cifar-10-batches-py/batches.meta")
label_names = metadata['label_names']

#he first 1024 entries contain the red channel values, the next 1024 the green,
#and the final 1024 the blue. The image is stored in row-major order, so that the
#first 32 entries of the array are the red channel values of the first row of the image.
data[0]

from google.colab.patches import cv2_imshow


fig=plt.figure(figsize=(10, 10))
for i in range(1, 26):
    fig.add_subplot(5, 5, i)
    img = np.reshape(data[i], (3,32,32)).transpose(1,2,0)
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
    plt.title(label_names[labels[i]])
plt.show()

def escalamiento(x):
    return -1 + 2/255*x

x = np.arange(0,256,1)
plt.title("Escalamiento lineal")
plt.plot(x,escalamiento(x))
plt.xlabel("Valor del pixel original")
plt.ylabel("Valor del pixel escalado")
plt.grid()



from torch.utils.data import Dataset

def unpickle(file):
```

```
96      with open(file, 'rb') as fo:
97          dict = pickle.load(fo, encoding='latin1')
98      return dict
99
100
101 class CIFAR10Train(Dataset):
102     def __init__(self, path):
103         # Constructor, debe leer el archivo data_batch_1 dentro de la carpeta
104         # indicada (este archivo se usará para el set de entrenamiento)
105         self.dict_data = unpickle(path+'data_batch_1')
106         self.labels,self.filenames = self.dict_data['labels'],self.dict_data['filenames']
107
108     def __len__(self):
109         # Debe retornar el número de imágenes en el dataset de entrenamiento
110         return len(self.filenames)
111
112     def __getitem__(self, index):
113         data = self.dict_data['data']
114         # Debe retornar un par label, image
115         # Donde label es una etiqueta, e image es un arreglo de 3x32x32
116         # index es un número (o lista de números) que indica cuáles imágenes
117         # y labels se deben retornar
118         #escalamineto lineal
119         datax = -1 +2/255*data[index]
120         img = np.reshape(datax, (3,32,32))
121         return self.labels[index], img
122
123 class CIFAR10Val(Dataset):
124     def __init__(self, path):
125         # Constructor, debe leer el archivo data_batch_2 dentro de la carpeta
126         # indicada (este archivo se usará para el set de entrenamiento)
127         self.dict_data = unpickle(path+'data_batch_2')
128         self.labels,self.filenames = self.dict_data['labels'],self.dict_data['filenames']
129
130     def __len__(self):
131         # Debe retornar el número de imágenes en el dataset de entrenamiento
132         return len(self.filenames)
133
134     def __getitem__(self, index):
135         data = self.dict_data['data']
136         # Debe retornar un par label, image
137         # Donde label es una etiqueta, e image es un arreglo de 3x32x32
138         # index es un número (o lista de números) que indica cuáles imágenes
139         # y labels se deben retornar
140         #escalamineto lineal
141         datax = -1 +2/255*data[index]
142         img = np.reshape(datax, (3,32,32))
143         return self.labels[index], img
144
145 class CIFAR10Test(Dataset):
146     def __init__(self, path):
```

```
147         # Constructor, debe leer el archivo test_batch
148         # indicada (este archivo se usará para el set de entrenamiento)
149         self.dict_data = unpickle(path+'test_batch')
150         self.labels,self.filenames = self.dict_data['labels'],self.dict_data['filenames']
151 
152     def __len__(self):
153         # Debe retornar el número de imágenes en el dataset de entrenamiento
154         return len(self.filenames)
155 
156     def __getitem__(self, index):
157         data = self.dict_data['data']
158         # Debe retornar un par label, image
159         # Donde label es una etiqueta, e image es un arreglo de 3x32x32
160         # index es un número (o lista de números) que indica cuáles imágenes
161         # y labels se deben retornar
162 
163         #escalamineto lineal
164         datax = -1 +2/255*data[index]
165         img = np.reshape(datax, (3,32,32))
166         return self.labels[index], img
167 
168 trainDataset = CIFAR10Train("/content/cifar-10-batches-py/")
169 valDataset = CIFAR10Val("/content/cifar-10-batches-py/")
170 testDataset = CIFAR10Test("/content/cifar-10-batches-py/")
171 
172 BATCH_SIZE = 256
173 train_loader = torch.utils.data.DataLoader(trainDataset, batch_size=BATCH_SIZE, shuffle=True,
        ↪ num_workers=4, pin_memory=True)
174 val_loader = torch.utils.data.DataLoader(valDataset, batch_size=BATCH_SIZE, shuffle=True,
        ↪ num_workers=4, pin_memory=True)
175 test_loader= torch.utils.data.DataLoader(testDataset, batch_size=BATCH_SIZE, shuffle=True,
        ↪ num_workers=4, pin_memory=True)
176 
177 """# Sección 2. Implementar una red inicial.
178 
179  Las primeras capas convolucionales que se recomienda usar están indicadas en el enunciado. Elija un
        ↪ tamaño N de neuronas en la última capa oculta que le parezca
180 apropiado. Se recomienda usar max pooling cada cierta cantidad de capas para reducir el tamaño
181 espacial de los tensores. Elija un batch_size inicial que le parezca apropiado.
182 """
183 
184 class MyNet(nn.Module):
185  def __init__(self, N=128):
186     super(MyNet, self).__init__()
187     self.nclasses = 10
188     #nn.Conv2d(in_channels, out_channels, kernel_size)
189     self.conv1 = nn.Conv2d(3, 64, 3, padding = 1) #64 filtros de 3x3, 3 canales de entrada
190     self.conv2 = nn.Conv2d(64, 64, 3, padding = 1)
191     self.conv3 = nn.Conv2d(64, 128, 3, padding = 1)
192 
193     self.bn1 = torch.nn.BatchNorm2d(64)
```

```
    self.bn2 = torch.nn.BatchNorm2d(64)
    self.bn3 = torch.nn.BatchNorm2d(128)

    self.MaxPool = nn.MaxPool2d(2, 2)

    self.fc1 = nn.Linear(8192, 1024)
    self.fc2 = nn.Linear(1024, 24*N)
    self.fc3 = nn.Linear(24*N, N)
    self.fc_last = nn.Linear(N, self.nclasses)
 def forward(self, x):
    x = self.bn1(F.relu(self.conv1(x)))
    x = self.MaxPool(self.bn2(F.relu(self.conv2(x))))
    x = self.MaxPool(self.bn3(F.relu(self.conv3(x))))

    #transformamos el tensor de una capa convolucional a una capa fully connected
    x = x.view(x.size()[0], -1)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = F.relu(self.fc3(x))
    x = self.fc_last(x)
    return x

"""## Sección 2.a Entrenar la red usando el conjunto de entrenamiento y controlando el sobreajuste
    ↪ con el conjunto de validación.


La red puede sufrir sobreajuste si la cantidad de parámetros es grande, a medida que el
    ↪ entrenamiento va
progresando. Para poder evitar el sobreajuste, se recomienda usar un enfoque basado en patience.
    ↪ Además,
se debe ir guardando el menor loss de validación, se debe ir guardando checkpoints cada vez que el
    ↪ loss
actual es menor que el menor loss de validación existente. Posteriormente, para poder evaluar el
desempeño de la red, se debe recuperar el mejor checkpoint almacenado
"""

# Parte de los prints de los accuracy's y de los loss se basaron en modelos
# del curso Deep learning que estoy cursando actualmente.
def train(net, optimizer, num_epocas):
  inicio = time.time()
  #copiamos el modelo utilizando la libreria copy
  best_model_wts = copy.deepcopy(net.state_dict())
  train_losses = []
  train_counter = []
  train_accuracy = []
  val_losses = []
  val_accuracy = []
  best_acc = 0.0
  best_loss = 2e32
  for epoch in range(num_epocas):
    print('Epoch {}/{}'.format(epoch, num_epocas-1))
```

```
241     print('-' * 10)
242 
243     net.train() #Modo entrenamiento
244 
245     running_loss = 0.0
246     running_corrects = 0.0
247     for i, data in enumerate(train_loader, 0): # Obtener batch
248         labels = data[0].cuda()
249         inputs = data[1].cuda().float()
250         optimizer.zero_grad()
251         outputs = net(inputs) #salidas de la red
252         preds = outputs.argmax(axis=1) #predicciones
253         loss = criterion(outputs, labels)
254         loss.backward()
255         optimizer.step()
256 
257         running_loss += loss.item() * inputs.size(0)
258         running_corrects += torch.sum(preds == labels.data)
259 
260     epoch_loss = running_loss /len(train_loader.dataset) #promedio de error
261     epoch_acc = running_corrects.double() / len(train_loader.dataset) #promedio de accuracy
262     train_losses.append(epoch_loss)
263     train_counter.append(epoch)
264     train_accuracy.append(epoch_acc)
265 
266     print('Train Loss: {:.4f} Acc: {:.4f}'.format(epoch_loss, epoch_acc))
267 
268     #Validacion
269     net.eval()
270 
271     running_loss = 0.0
272     running_corrects = 0.0
273     for labels,inputs in val_loader:
274         inputs = inputs.to(device).float()
275         labels = labels.to(device)
276         with torch.set_grad_enabled(False):
277             outputs = net(inputs)
278             preds = outputs.argmax(axis=1)
279             val_loss = criterion(outputs, labels)
280             #val_losses.append(val_loss.item())
281             #correct += pred.eq(target.data.view_as(pred)).sum()
282 
283             running_loss += loss.item() * inputs.size(0)
284             running_corrects += torch.sum(preds == labels.data)
285 
286     epoch_loss = running_loss /len(val_loader.dataset)
287     epoch_acc = running_corrects.double() / len(val_loader.dataset)
288     val_losses.append(epoch_loss)
289     val_accuracy.append(epoch_acc)
290     print('Val Loss: {:.4f} Acc: {:.4f}'.format(epoch_loss, epoch_acc))
291 
```

```
292     #chekpoint
293     if epoch_loss < best_loss:
294         best_loss = epoch_loss
295         best_model_wts = copy.deepcopy(net.state_dict())
296 
297     # early stopping, si el error aumenta más de 5 veces respecto al menor error,
298     # terminamos el entrenamiento
299     if epoch_loss > best_loss*5:
300         print('\n'+'-' * 10+'Early Stopping'+'-' * 10+'\n')
301         break
302 
303   print('Best val loss: {:.4f}'.format(best_loss))
304   plt.figure()
305   #2b. Graficar las curvas de loss de entrenamiento y validación
306   plt.title("Error en cada epoca")
307   plt.plot(train_counter, train_losses, label='Entrenamiento',color='blue')
308   plt.plot(train_counter,val_losses, label='Validacion',color='red')
309   plt.xlabel("Epochs")
310   plt.ylabel("Loss")
311   plt.legend()
312   plt.show()
313 
314   final = time.time()
315   print('Training complete in {:.0f}m {:.0f}s'.format((final-inicio)//60, (final-inicio) % 60))
316 
317   net.load_state_dict(best_model_wts)
318   return net
319 
320 from sklearn.metrics import confusion_matrix
321 import matplotlib.pyplot as plt
322 import itertools
323 
324 
325 def plot_confusion_matrix(cm, classes,accuracy,N,
326                           title='Confusion matrix',
327                           cmap=plt.cm.Blues):
328 
329 
330   cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
331   plt.figure(figsize=(10,7))
332   plt.imshow(cm, interpolation='nearest', cmap=cmap)
333   plt.title(title+'\n Accuracy = '+str(round(accuracy,2))+' %'+' N = '+str(N))
334   plt.colorbar()
335   tick_marks = np.arange(len(classes))
336   plt.xticks(tick_marks, classes, rotation=45)
337   plt.yticks(tick_marks, classes)
338 
339   fmt = '.2f'
340   thresh = cm.max() / 2.
341   for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
342       plt.text(j, i, format(cm[i, j], fmt),
```

```
            horizontalalignment="center",
            color="white" if cm[i, j] > thresh else "black")

  plt.tight_layout()
  plt.ylabel('True label')
  plt.xlabel('Predicted label')
  plt.show()

# calculate accuracy
from sklearn.metrics import  accuracy_score

def evaluar_red(best_net,N,plot=True):
    #Evaluamos la red con los conjuntos de entrenamiento y validación
    best_net.eval()

    y_pred = []
    y_train = []
    for labels,inputs in train_loader:
        inputs = inputs.to(device).float()
        labels = labels.to(device)
        y_train += labels.cpu().tolist()
        with torch.no_grad():
            outputs = best_net(inputs)
            preds = outputs.argmax(axis=1)
            y_pred += preds.cpu().tolist()

    accuracy = accuracy_score(y_train, y_pred)*100
    cm = confusion_matrix(y_train, y_pred)
    if plot:
        plot_confusion_matrix(cm, list(range(10)), accuracy,N, title="Matriz de confusión Conjunto
     ↪ Entrenamiento")

    #Val
    best_net.eval()
    y_pred = []
    y_val = []
    for labels,inputs in val_loader:
        inputs = inputs.to(device).float()
        labels = labels.to(device)
        y_val += labels.cpu().tolist()
        with torch.no_grad():
            outputs = best_net(inputs)
            preds = outputs.argmax(axis=1)
            y_pred += preds.cpu().tolist()

    accuracy = accuracy_score(y_val, y_pred)*100
    cm = confusion_matrix(y_val, y_pred)
    if plot:
        plot_confusion_matrix(cm, list(range(10)), accuracy,N, title="Matriz de confusión Conjunto
     ↪ Validación")
    #retornamos el accuracy en el conjunto de validación
```

```
392     return accuracy
393 
394 """Inicializamos la red."""
395 
396 N = 256
397 net = MyNet(N)
398 net.cuda()
399 criterion = nn.CrossEntropyLoss()
400 optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
401 
402 """## Sección 2a y 2b. Entrenar y graficar las curvas de loss de entrenamiento y validación"""
403 
404 best_net_1 = train(net, optimizer, num_epocas=15)
405 
406 """# Sección 2.c Evaluar la red sobre los conjuntos de entrenamiento y validación, usando el mejor
        ↪ checkpoint almacenado"""
407 
408 evaluar_red(best_net_1,N=256)
409 
410 """# Sección 3. Modificar el valor de N, repitiendo el Paso 2 hasta obtener una red con un buen
        ↪ desempeño.
411 
412 Se prueban distintos valores de N, buscando obtener una red con mejor desempeño. Se selecciona el
        ↪ N que arroja mayor accuracy en el conjunto de validación.
413 """
414 
415 models = []
416 accuracys = []
417 n_list = [16,32,64,128,256]
418 for N in n_list:
419     print(f'\nModel con N={N}')
420     net = MyNet(N)
421     net.cuda()
422     criterion = nn.CrossEntropyLoss()
423     optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
424     best_net = train(net, optimizer, num_epocas=20)
425     models.append(best_net)
426     accuracys.append(evaluar_red(best_net,N,plot=False))
427 
428 """## Resultados"""
429 
430 plt.plot(n_list,accuracys,'o-')
431 plt.title('Accuracy validación vs N ')
432 plt.ylabel('Accuracy validación')
433 plt.xlabel('N (Neuronas última capa oculta)')
434 plt.grid()
435 # Texto en la gráfica en coordenadas (x,y)
436 idx = np.argmax(accuracys)
437 
438 texto1 = plt.text( n_list[idx]+15,accuracys[idx]-0.05, f'N={n_list[idx]}, A='+str(round(accuracys[
        ↪ idx],2))+' %', fontsize=10)
```

```
439
440 N = n_list[idx]
441
442 #evaluamos el desempeño con el modelo que entrega mejor accuracy de validación
443 evaluar_red(models[idx],n_list[idx])
444
445 """# Sección 4. Repetir el Paso 2 usando dos números distintos de capas convolucionales y elija el
        ↪ que genere los mejores resultados"""
446
447 # agregando una capa convolucional más
448 class MyNet2(nn.Module):
449  def ___init___(self, N=128):
450    super(MyNet2, self).___init___()
451    self.nclasses = 10
452    #nn.Conv2d(in_channels, out_channels, kernel_size)
453    self.conv1 = nn.Conv2d(3, 64, 3, padding = 1) #64 filtros de 3x3, 3 canales de entrada
454    self.conv2 = nn.Conv2d(64, 64, 3, padding = 1)
455    self.conv3 = nn.Conv2d(64, 128, 3, padding = 1)
456    self.conv4 = nn.Conv2d(128, 256, 3, padding = 1)
457
458    self.bn1 = torch.nn.BatchNorm2d(64)
459    self.bn2 = torch.nn.BatchNorm2d(64)
460    self.bn3 = torch.nn.BatchNorm2d(128)
461    self.bn4 = torch.nn.BatchNorm2d(256)
462
463    self.MaxPool = nn.MaxPool2d(2, 2)
464
465    self.fc1 = nn.Linear(4096, 1024)
466    self.fc2 = nn.Linear(1024, 24*N)
467    self.fc3 = nn.Linear(24*N, N)
468    self.fc_last = nn.Linear(N, self.nclasses)
469  def forward(self, x):
470    x = self.bn1(F.relu(self.conv1(x)))
471    x = self.MaxPool(self.bn2(F.relu(self.conv2(x))))
472    x = self.MaxPool(self.bn3(F.relu(self.conv3(x))))
473    x = self.MaxPool(self.bn4(F.relu(self.conv4(x))))
474
475    #transformamos el tensor de una capa convolucional a una capa fully connected
476    x = x.view(x.size()[0], -1)
477    x = F.relu(self.fc1(x))
478    x = F.relu(self.fc2(x))
479    x = F.relu(self.fc3(x))
480    x = self.fc_last(x)
481    return x
482
483 print(f'\nModel con N={N}')
484 net3 = MyNet2(N)
485 net3.cuda()
486 criterion = nn.CrossEntropyLoss()
487 optimizer = torch.optim.Adam(net3.parameters(), lr=1e-3)
488 best_net3 = train(net3, optimizer, num_epocas=20)
```

```
489
490 evaluar_red(best_net3,N)
491
492 #agregando 2 capas convolucionales más
493 class MyNet3(nn.Module):
494  def ___init___(self, N=16):
495    super(MyNet3, self).___init___()
496    self.nclasses = 10
497    #nn.Conv2d(in_channels, out_channels, kernel_size)
498    self.conv1 = nn.Conv2d(3, 64, 3, padding = 1) #64 filtros de 3x3, 3 canales de entrada
499    self.conv2 = nn.Conv2d(64, 64, 3, padding = 1)
500    self.conv3 = nn.Conv2d(64, 128, 3, padding = 1)
501    self.conv4 = nn.Conv2d(128, 256, 3, padding = 1)
502    self.conv5 = nn.Conv2d(256, 512, 3, padding = 1)
503
504    self.bn1 = torch.nn.BatchNorm2d(64)
505    self.bn2 = torch.nn.BatchNorm2d(64)
506    self.bn3 = torch.nn.BatchNorm2d(128)
507    self.bn4 = torch.nn.BatchNorm2d(256)
508    self.bn5 = torch.nn.BatchNorm2d(512)
509
510    self.MaxPool = nn.MaxPool2d(2, 2)
511
512    self.fc1 = nn.Linear(8192, 1024)
513    self.fc2 = nn.Linear(1024, 24*N)
514    self.fc3 = nn.Linear(24*N, N)
515    self.fc_last = nn.Linear(N, self.nclasses)
516  def forward(self, x):
517    x = self.bn1(F.relu(self.conv1(x)))
518    x = self.MaxPool(self.bn2(F.relu(self.conv2(x))))
519    x = self.MaxPool(self.bn3(F.relu(self.conv3(x))))
520    x = self.bn4(F.relu(self.conv4(x)))
521    x = self.MaxPool(self.bn5(F.relu(self.conv5(x))))
522
523    #transformamos el tensor de una capa convolucional a una capa fully connected
524    x = x.view(x.size()[0], -1)
525    x = F.relu(self.fc1(x))
526    x = F.relu(self.fc2(x))
527    x = F.relu(self.fc3(x))
528    x = self.fc_last(x)
529    return x
530
531 print(f'\nModel con N={N}')
532 net4 = MyNet3(N)
533 net4.cuda()
534 criterion = nn.CrossEntropyLoss()
535 optimizer = torch.optim.Adam(net4.parameters(), lr=1e-3)
536 best_net4 = train(net4, optimizer, num_epocas=20)
537
538 evaluar_red(best_net4,N)
539
```

```
"""# Sección 5. Usando la mejor configuración obtenida en los pasos anteriores, evaluar la mejor red sobre el conjunto de prueba."""

# calculate accuracy
from sklearn.metrics import  accuracy_score

#Evaluamos la red con los conjuntos de prueba
best_net4.eval()
y_pred = []
y_test = []
for labels,inputs in test_loader:
    inputs = inputs.to(device).float()
    labels = labels.to(device)
    y_test += labels.cpu().tolist()
    with torch.no_grad():
        outputs = best_net4(inputs)
        preds = outputs.argmax(axis=1)
        y_pred += preds.cpu().tolist()

accuracy = accuracy_score(y_test, y_pred)*100
cm = confusion_matrix(y_test, y_pred)
plot_confusion_matrix(cm, list(range(10)), accuracy,N, title="Matriz de confusión Conjunto Prueba" )
```