1st Summer School Physical Engineering UNI 2022-Lima,Perú

Deep Learning Report

Becerra Ahumada, Carlo Eduardo[1]

[1]Facultad de Ciencias, Universidad Nacional de Ingenieria
[1]cbecerraa@uni.pe

February 28, 2022

Resumen

Este trabajo consiste en la clasificación de un conjunto imágenes mediante el uso de redes neuronales usando el entorno de PyTorch. Se procedió a entrenar la red neuronal con ocho mil imágenes de perros y gatos en formato RGB, para posteriormente testear dos mil datos de otras imágenes de perros y gatos, también en formato RGB, con lo aprendido. Se procedió también a usar dos optimizadores, el *SGD*, y el optimizador *Adam*; así como también el uso de shuffle = True y shuffle = False para cada optimizador.

Palabras Clave:
Redes neuronales, computer vision, deep learning, image classification, clasificación de imágenes.

1 Metodología

Para el desarrollo de este trabajo, se usaron funciones de la librería PyTorch, para la implementación de nuestra pequeña red neuronal; entre ellas, destacan las principales funciones que usamos al momento de construir nuestro modelo, que enumeraremos a continuación.

- *nn.Conv2d:* Esta función, nos ayuda a detectar características relevantes de las imágenes a analizar; hace que nuestra imagen sea cada vez más pequeña, al detectar tales características, por lo que va disminuyendo de tamaño. Los parámetros de esta función son los siguientes:

$$\begin{aligned} nn.Conv2d(&\#canales\ entrada,\\ &\#canales\ salida,\\ &kernel,\\ &bias = True\ or\ False) \end{aligned} \quad (1)$$

  recibe como parámetros de entrada, al número de canales que tiene la imagen que vamos a convolucionar, el siguiente parámetro es el número de canales que quermos que nos entregue, el tercer parámetro es el tamaño del kernel que se va a aplicar al realizar la convolución; y por último, al menos en esta ocasión, se tiene al bias, si queremos usar un bias al momento de realizar la convolución, se le otorga el valor de *True*, de lo contrario, se le da el valor de *False*.

- *nn.BatchNorm2d:* Esta función es una especie de reescalamiento de los datos que se tiene, que evita que los valores no crezcan de manera descontrolada. Tiene como parámetro de entrada el número de canales de la imagen a usar.

- *ReLU:* Esta es la función de activación que estamos usando, sirve para que se activen las *neuronas* de la red.

- *nn.MaxPool2d:* Extrae el máximo valor de una determinada sección de la imagen a analizar.

Asimismo, también tenemos otras funciones al momento de introducir nuestros datos al momento del entrenamiento de la red, tales como las pertenecientes a la librería *torchvision.transforms.transforms*; entre las que tenemos:

- *RandomRotation:* que como indica su nombre, rota aleatoriamente a alguna de las imágenes de nuestro conjunto de datos, la cantidad de grados que deseemos.

- *RandomHorizontalFlip:* Realiza la operación de reflejar la imagen respecto al eje vertical de uno de los extremos verticales.

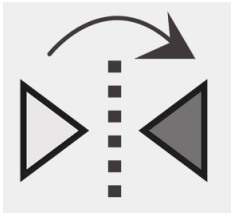

Figura 1: RandomHorizontalFlip

- *Resize():* Dado que no todos nuestras imágenes son del mismo tamaño, con esta función podemos uniformizarlas, insertando como parámetros de entrada las dimensiones de alto y ancho que deseamos.
- *ToTensor():* Con esta función, transformamos los valores de la imagen a un tensor, para su posterior uso al momento de usar las funciones ya mencionadas en la construcción del modelo.

Tenemos también los optimizadores que se usan para minimizar la función loss del proceso, esto son los optimizadores de *SGD*, que usa el gradiente, tratando de dirigirse hacia donde el valor del gradiente sea menor; y el optimizador Adam, cuyo funcionamiento es más complicado de explicar, pero que también busca minimizar la función loss del proceso a realizar.

2 Procedimiento

En esta sección, se detallan las partes del código implementado.

2.1 Librerías

Para el funcionamiento óptimo de este proyecto, se importaron las siguientes librerías:

```
import torch
import torch.nn as nn
import torchvision
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import numpy as np
import pandas as pd
from collections import defaultdict
import os
import PIL
```

donde, estamos importando librerías de redes neuronales y las clásicas libreías para *plotear* imágenes.

2.2 Usando el GPU

Definimos el dispositivo donde trabajeremos, en este caso usaremos la GPU que proporciona *Google Colab*

```
dev = torch.device("cuda:0" if torch.cuda.is_available()
else "cpu")
```

2.3 Extracción de datos

Para ello, hemos de crear una cuenta en *kaggle.com*; descargar nuestro API Token (*kaggle.json*), subirlo al notebook de *Google Colab* para así poder descargar la base de datos. Mediante la siguiente línea, creamos una carpeta .kaggle y movemos el archivo recién subido ahí:

```
!mkdir .kaggle
!mv kaggle.json .kaggle/
!mv .kaggle ~/
```

Extraemos el archivo que queremos, en este caso usamos el dataset de perros y gatos

```
!kaggle datasets download chetankv/dogs-cats-images
!unzip dogs-cats-images.zip
```

3 Resultados

3.1 Funciones que entrenan al modelo

En este apartado, se colocan las funciones que ayudan al entrenamiento de la red neuronal, aplicando una serie de procedimientos a los datos de las imágenes a usar

```
def evaluate(model, loader, crit):
  model.eval()
  total = 0
  corrects = 0
  avg_loss = 0
  for x, y in loader:
    x = x.to(dev)
    y = y.to(dev)
    o = model(x)
    loss = crit(o,y)
    avg_loss += loss.item()
    corrects += torch.sum(torch.argmax(o,axis=1)
    == y).item()
    total += len(y)
  acc = 100* corrects / total
  avg_loss /= len(loader)
  return avg_loss, acc

def train_one_epoch(model, train_loader, crit, optim):
  model.train()
  total = 0
  corrects = 0
  avg_loss = 0
  for x, y in train_loader:
    optim.zero_grad()
    x = x.to(dev)
    y = y.to(dev)
    o = model(x)
    loss = crit(o,y)
    avg_loss += loss.item()
    loss.backward()
    optim.step()
    corrects += torch.sum(torch.argmax(o,axis=1) == y)
    .item()
    total += len(y)
  acc = 100 * corrects / total
  avg_loss /= len(train_loader)
  return avg_loss, acc

arr_train_loss = []
arr_test_loss = []
arr_train_acc = []
arr_test_acc = []
def train(model, train_loader, test_loader, crit,
optim, epochs = 20):
  for epoch in range(epochs):
    train_loss, train_acc = train_one_epoch(model,
    train_loader,crit, optim)
    test_loss, test_acc = evaluate(model, test_loader,
    crit)
    print(f"epoch: {epoch}, train loss: {train_loss},
    train acc: {train_acc}%, test loss: {test_loss},
    test acc: {test_acc}%")
```

```
    arr_train_loss.append(train_loss)
    arr_test_loss.append(test_loss)
    arr_train_acc.append(train_acc)
    arr_test_acc.append(test_acc)
```

En este caso, se añadieron cuatro *arrays* para el almacenamiento de los valores del **train loss, train acc, test loss, test acc**, para graficarlos posteriormente.

3.2 Transformación de los datos

Dado que las imágenes vienen en bytes, tenemos que adaptar estos valores a que sean parte de un tensor, pero también con la finalidad de enriquecer un poco los datos de entrenamiento usamos las funciones ya mencionadas en la Sec. 1, tales como la rotación de la imagen, el redimensiomamiento de las mismas, y su reflexión horizontal respecto a un eje vertical. Ello se realiza con las siguientes líneas de código.

```
from torchvision.transforms.transforms
import RandomRotation
img_transform = torchvision.transforms.Compose([
  torchvision.transforms.RandomRotation(2),
  #Rotamos aleatoriamente algunas imágenes 2°


  torchvision.transforms.RandomHorizontalFlip(),
  # Con esta línea de código, rotamos aleatoriamente
  alguna imagen horizontalmente


  torchvision.transforms.Resize((256,256)),
  # Con esta línea uniformizamos los tamaños de todas
  las imágenes


  torchvision.transforms.ToTensor()
  # con esta línea, transfomramos los valores de
  la imagen a un tensor
])
```

3.3 Dirección donde se encuentran los datos

Ponemos la ubicación de los datos en el notebook de *Google Colab*, y denotamos los archivos que servirán para el entrenamiento de la red, así como los archivos que se usarán para el testeo de la misma.

```
train_ds = torchvision.datasets.ImageFolder("/content/
dataset/training_set",transform=img_transform)
test_ds = torchvision.datasets.ImageFolder("/content/
dataset/test_set",transform=img_transform)
```

3.4 Muestra de imagen a usar

Se muestra una de las tantas imágenes que se usan para el entrenamiento.

```
plt.imshow(train_ds[6531][0].numpy().transpose(1,2,0))
```

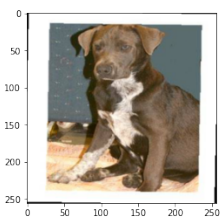

Figura 2: Imagen muestral

La imagen se puede ver en la Fig. 2.

3.5 Agrupación de datos

Como el conjunto de datos es demasiado grande, se forman bloques de cantidades más pequeñas, en este caso, de 128 imágenes.

```
# Formamos paquetes de 128 datos, y colocamos shuffle=True,
esto con el fin que el aprendizaje de la red neuronal se
vaya almacenando, y que no se vaya olvidando de lo que ha
aprendido
train_dl = torch.utils.data.DataLoader(train_ds,batch_size=128,
shuffle=True)
test_dl = torch.utils.data.DataLoader(test_ds,batch_size=128,
shuffle=True)
```

3.6 Modelo usado

Se muestra el código del modelo usado

```
model = nn.Sequential(
  nn.Conv2d(3,16,4,bias=False),
  # Nos ayuda a detectar los bordes y
  características distintivas de la imagen.


  nn.BatchNorm2d(16),
  # Es un reescalamiento de los datos,
  evita que los valores crezcan descontroladamente.


  nn.ReLU(inplace=True),
  # Función de activación.


  nn.MaxPool2d(2),
  # EScoge el máximo, en un kernel de 2x2.

  nn.Conv2d(16,32,3,bias=False),
  nn.BatchNorm2d(32),
  nn.ReLU(inplace=True),
  nn.MaxPool2d(2),

  nn.Conv2d(32,32,2,bias=False),
  nn.BatchNorm2d(32),
  nn.ReLU(inplace=True),
  nn.MaxPool2d(2),

  nn.Conv2d(32,64,2,bias=False),
  nn.BatchNorm2d(64),
  nn.ReLU(inplace=True),
  nn.MaxPool2d(2),

  nn.Conv2d(64,64,3,bias=False),
  nn.BatchNorm2d(64),
  nn.ReLU(inplace=True),
  nn.MaxPool2d(2),

  nn.Conv2d(64,128,3,bias=False),
  nn.BatchNorm2d(128),
  nn.ReLU(inplace=True),
  nn.MaxPool2d(2),

  nn.Flatten(),
  # Aplana el tensor en dos dimensiones

  nn.Linear(512,2)
  # Aplica una transformación lineal;
  el 2, indica la cantidad de salidas
  que queremos extraer, en este caso,
  se quieren dos salidas, o perros, o gatos.
).to(dev)
```

En este apartado, se construyó el modelo que se aplicó a nuestro problema, para ello, usamos las funciones de nn.Conv2d reiteradas veces, así como nn.BatchNorm2d, nn.ReLU, nn.MaxPool2d, siempre teniendo en cuenta cómo iban evolucionando los datos de entrada, para la extracción de características de nuestras

imágenes. En la parte final se usó la función nn.Flatten para *aplanar* al tensor en dos dimensiones y finalmente se usó una transformación lineal esperando solo dos parámetros de salida, donde el valor de 0 indica que se trata de un gato, y el valor de 1 indica que se trata de un perro.

3.7 Entrenamiento de la Red neuronal

En este apartado, indicamos la función de Loss que se usa, en este caso es la nn.CrossEntropyLoss; además se indica el optimizador que se quiere usar (que variaremos entre SGD y Adam) y el learning ratio a usar.

```
crit = nn.CrossEntropyLoss()
# Definimos el Lost

optim = torch.optim.SGD(model.parameters(),lr=0.1)
# Definimos el optimizador

train(model,train_dl, test_dl, crit, optim, epochs=10)
# Entrenamos el modelo
```

3.8 Testeo

En esta última parte, vemos los resultados que nuestra red entrenada puede predecir.

```
model.eval()
idx = 1000
x, y = test_ds[idx]
x_numpy = x.numpy().transpose(1,2,0)
N, H, W = x.shape
x = x.reshape(1,N,H,W)
pred = torch.argmax(model(x.to(dev)).cpu())
.item()
if pred==0:
  print('Gato')
if pred==1:
  print('Perro')
plt.imshow(x_numpy)
```

4 Resultados

4.1 SGD, shuffle = True

- Para la imagen de testeo 1000, se obtuvo una clasificación exitosa.

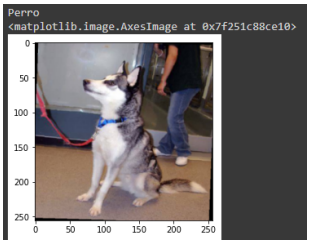

Figura 3: SGD, shuffle = TRUE; imagen 1000

- Para la imagen de testeo 1030, se obtuvo una clasificación exitosa.

Figura 4: SGD, shuffle = TRUE; imagen 1030

- Para la imagen de testeo 1997, se obtuvo una clasificación errónea.

Figura 5: SGD, shuffle = TRUE; imagen 1997

- Para la imagen de testeo 500, se obtuvo una clasificación exitosa.

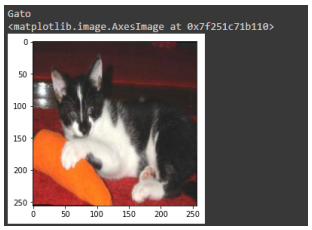

Figura 6: SGD, shuffle = TRUE; imagen 500

- Para la imagen de testeo 667, se obtuvo una clasificación exitosa.

Figura 7: SGD, shuffle = TRUE; imagen 667

- Para la imagen de testeo 1577, se obtuvo una clasificación exitosa.

Figura 8: SGD, shuffle = TRUE; imagen 1577

- Para la imagen de testeo 1234, se obtuvo una clasificación exitosa.

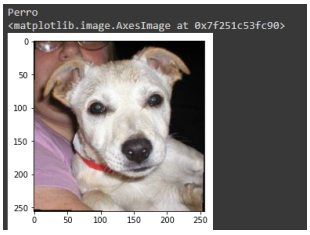

Figura 9: SGD, shuffle = TRUE; imagen 1234

- Para la imagen de testeo 1576, se obtuvo una clasificación errónea.

Figura 10: SGD, shuffle = TRUE; imagen 1576

Se obtuvo además la evolución del train loss, train acc, test loss, test acc; esto se muestra en la Fig. 11.

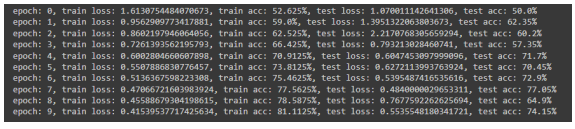

Figura 11: Evolución de los valores de **train loss, train acc, test loss, test acc**, para SGD, shuffle= True

Graficando dichos valores, se obtienen los gráficos de $train_loss$ vs $test_loss$ en la Fig. 12; y de $train_acc$ vs $test_acc$ en la Fig. 13.

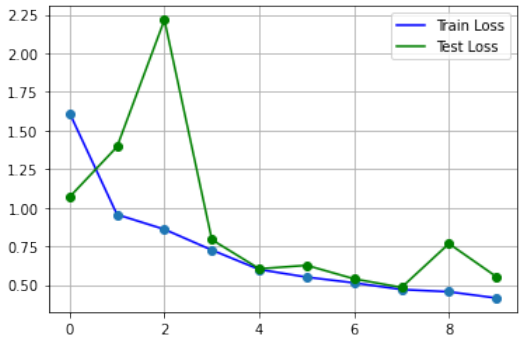

Figura 12: *train_loss* vs *test_loss*, para SGD, shuffle= True

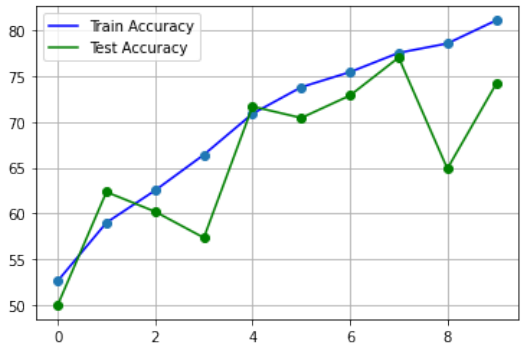

Figura 13: *train_acc* vs *test_acc*, para SGD, shuffle= True

4.2 SGD, shuffle = False

- Para la imagen de testeo 1000, se obtuvo una clasificación exitosa.

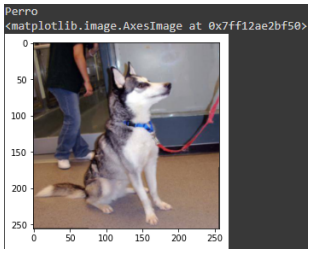

Figura 14: SGD, shuffle = FALSE; imagen 1000

- Para la imagen de testeo 1030, se obtuvo una clasificación exitosa.

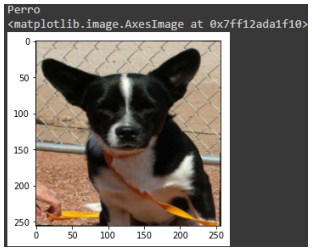

Figura 15: SGD, shuffle = FALSE; imagen 1030

- Para la imagen de testeo 1997, se obtuvo una clasificación exitosa.

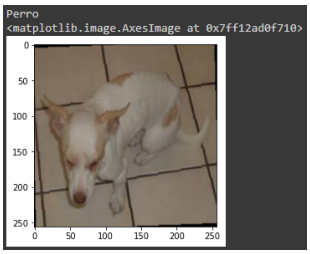

Figura 16: SGD, shuffle = FALSE; imagen 1997

- Para la imagen de testeo 500, se obtuvo una clasificación errónea.

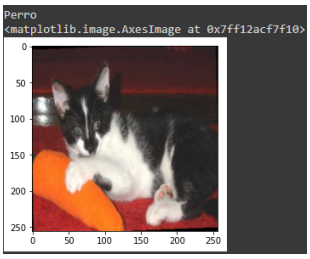

Figura 17: SGD, shuffle = FALSE; imagen 500

- Para la imagen de testeo 667, se obtuvo una clasificación errónea.

Figura 18: SGD, shuffle = FALSE; imagen 667

- Para la imagen de testeo 1577, se obtuvo una clasificación exitosa.

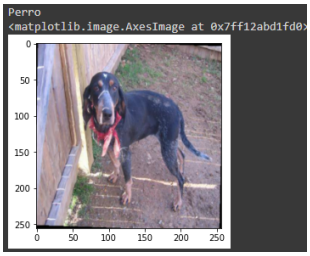

Figura 19: SGD, shuffle = FALSE; imagen 1577

- Para la imagen de testeo 1234, se obtuvo una clasificación exitosa.

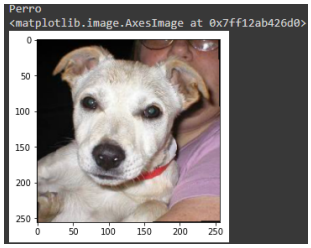

Figura 20: SGD, shuffle = FALSE; imagen 1234

- Para la imagen de testeo 1576, se obtuvo una clasificación exitosa.

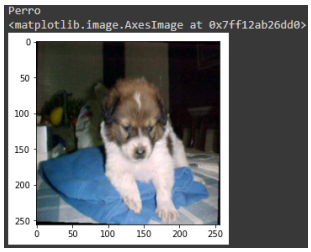

Figura 21: SGD, shuffle = FALSE; imagen 1576

Se obtuvo además la evolución del train loss, train acc, test loss, test acc; esto se muestra en la Fig. 22.

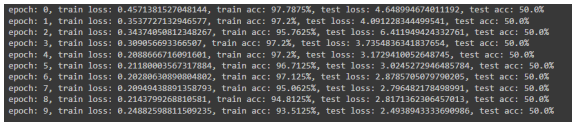

Figura 22: Evolución de los valores de **train loss, train acc, test loss, test acc**, para SGD, shuffle= False

Graficando dichos valores, se obtienen los gráficos de $train_loss$ vs $test_loss$ en la Fig. 23; y de $train_acc$ vs $test_acc$ en la Fig. 24.

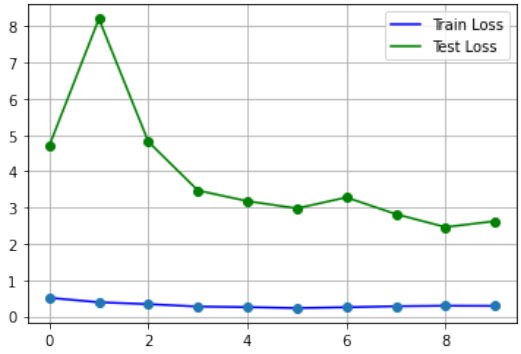

Figura 23: *train_loss* vs *test_loss*, para SGD, shuffle= False

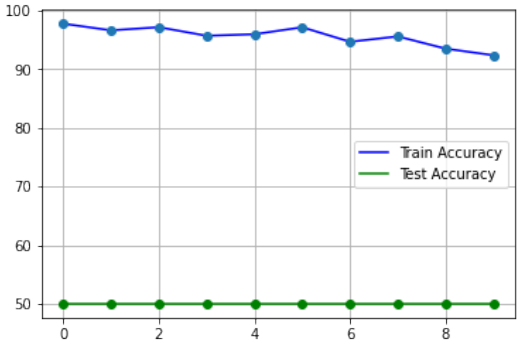

Figura 24: *train_acc* vs *test_acc*, para SGD, shuffle= False

4.3 Adam, shuffle = True

- Para la imagen de testeo 1000, se obtuvo una clasificación errónea.

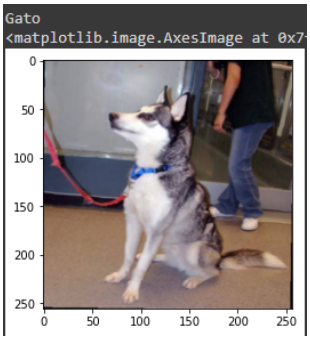

Figura 25: Adam, shuffle = TRUE; imagen 1000

- Para la imagen de testeo 1030, se obtuvo una clasificación errónea.

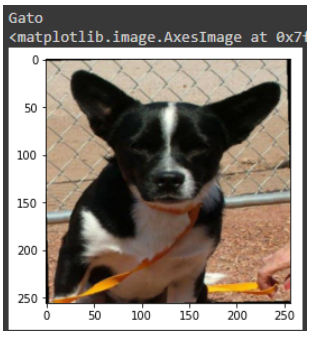

Figura 26: Adam, shuffle = TRUE; imagen 1030

- Para la imagen de testeo 1997, se obtuvo una clasificación errónea.

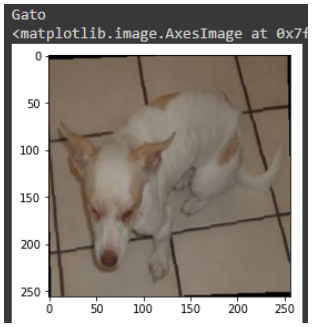

Figura 27: Adam, shuffle = TRUE; imagen 1997

- Para la imagen de testeo 500, se obtuvo una clasificación exitosa.

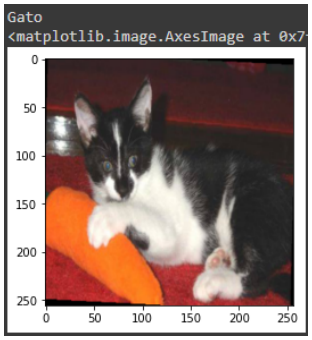

Figura 28: Adam, shuffle = TRUE; imagen 500

- Para la imagen de testeo 667, se obtuvo una clasificación exitosa.

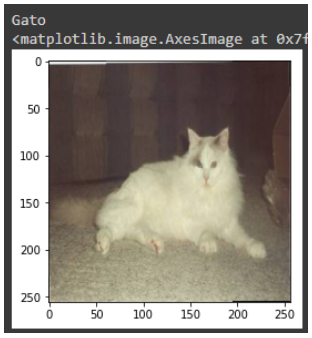

Figura 29: Adam, shuffle = TRUE; imagen 667

- Para la imagen de testeo 1577, se obtuvo una clasificación exitosa.

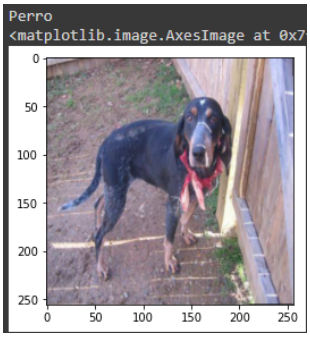

Figura 30: Adam, shuffle = TRUE; imagen 1577

- Para la imagen de testeo 1234, se obtuvo una clasificación exitosa.

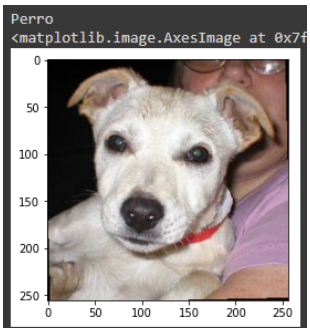

Figura 31: Adam, shuffle = TRUE; imagen 1234

- Para la imagen de testeo 1576, se obtuvo una clasificación errónea.

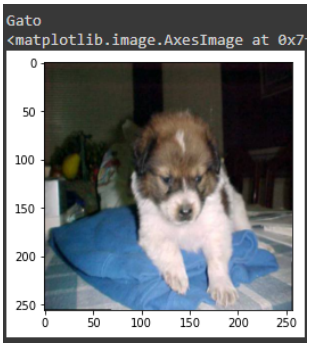

Figura 32: Adam, shuffle = TRUE; imagen 1576

Se obtuvo además la evolución del train loss, train acc, test loss, test acc; esto se muestra en la Fig. 11.

```
epoch: 0, train loss: 0.7707743990307763, train acc: 55.95%, test loss: 0.6441076062619686, test acc: 64.15%
epoch: 1, train loss: 0.6270484758747948, train acc: 65.6375%, test loss: 0.6152208186686039, test acc: 66.85%
epoch: 2, train loss: 0.5780096002041347, train acc: 69.875%, test loss: 0.5793720446527004, test acc: 70.9%
epoch: 3, train loss: 0.5241599712106917, train acc: 74.1625%, test loss: 0.6599201075732708, test acc: 68.7%
epoch: 4, train loss: 0.4863238868632637, train acc: 76.0875%, test loss: 0.4796734545170877, test acc: 76.0%
epoch: 5, train loss: 0.4339741544118003, train acc: 79.55%, test loss: 0.7128028869628906, test acc: 63.75%
epoch: 6, train loss: 0.4001750775745937, train acc: 81.2875%, test loss: 0.445867085829772, test acc: 78.1%
epoch: 7, train loss: 0.3575029155564687, train acc: 84.025%, test loss: 0.5472804494202137, test acc: 74.0%
epoch: 8, train loss: 0.3218164443697266, train acc: 86.05%, test loss: 0.7921109050512314, test acc: 65.65%
epoch: 9, train loss: 0.2913500062797893, train acc: 87.4875%, test loss: 0.3673803601413965, test acc: 84.55%
```

Figura 33: Evolución de los valores de **train loss, train acc, test loss, test acc**, para Adam, shuffle= True

Graficando dichos valores, se obtienen los gráficos de $train_loss$ vs $test_loss$ en la Fig. 34; y de $train_acc$ vs $test_acc$ en la Fig. 35.

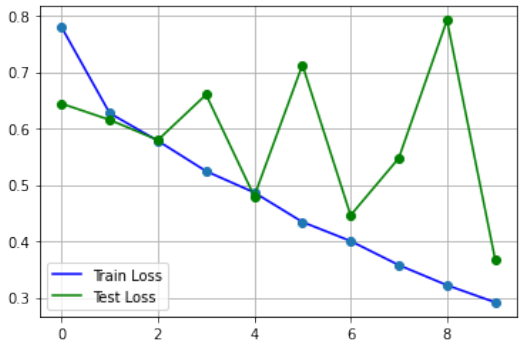

Figura 34: *train_loss* vs *test_loss*, para Adam, shuffle= True

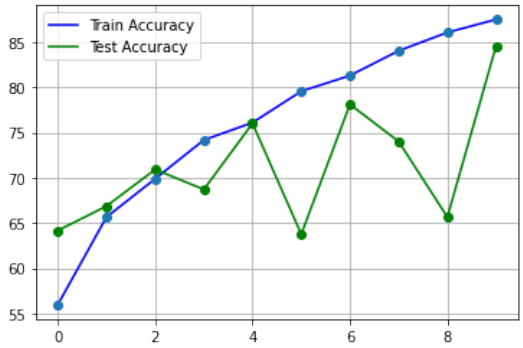

Figura 35: *train_acc* vs *test_acc*, para Adam, shuffle= True

4.4 Adam, shuffle = False

- Para la imagen de testeo 1000, se obtuvo una clasificación exitosa.

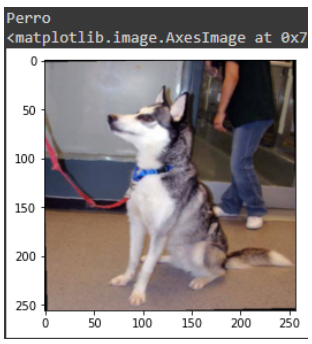

Figura 36: Adam, shuffle = FALSE; imagen 1000

- Para la imagen de testeo 1030, se obtuvo una clasificación exitosa.

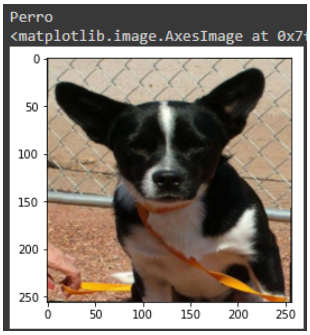

Figura 37: Adam, shuffle = FALSE; imagen 1030

- Para la imagen de testeo 1997, se obtuvo una clasificación exitosa.

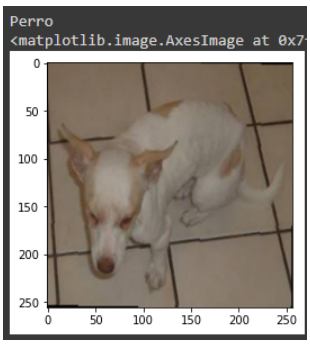

Figura 38: Adam, shuffle = FALSE; imagen 1997

- Para la imagen de testeo 500, se obtuvo una clasificación errónea.

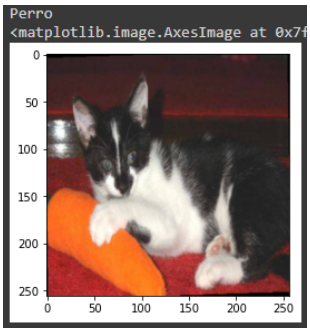

Figura 39: Adam, shuffle = FALSE; imagen 500

- Para la imagen de testeo 667, se obtuvo una clasificación errónea.

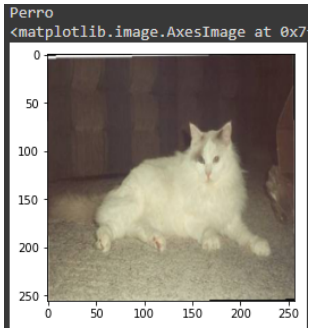

Figura 40: Adam, shuffle = FALSE; imagen 667

- Para la imagen de testeo 1577, se obtuvo una clasificación exitosa.

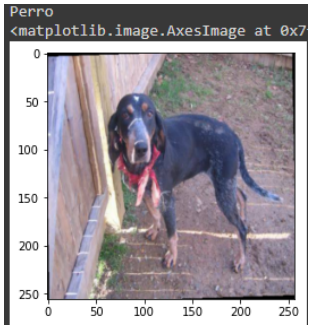

Figura 41: Adam, shuffle = FALSE; imagen 1577

- Para la imagen de testeo 1234, se obtuvo una clasificación exitosa.

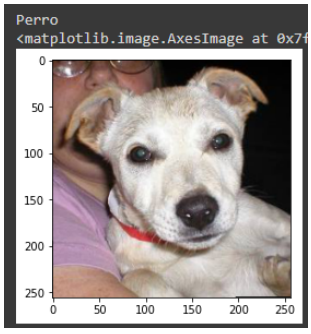

Figura 42: Adam, shuffle = FALSE; imagen 1234

- Para la imagen de testeo 1576, se obtuvo una clasificación exitosa.

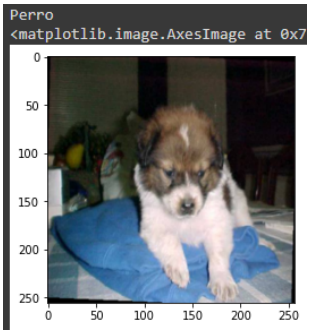

Figura 43: Adam, shuffle = FALSE; imagen 1576

Se obtuvo además la evolución del train loss, train acc, test loss, test acc; esto se muestra en la Fig. 11.

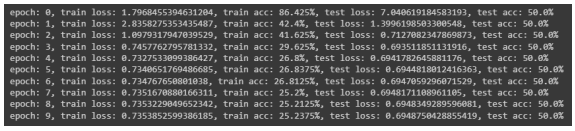

Figura 44: Evolución de los valores de **train loss, train acc, test loss, test acc**, para Adam, shuffle= False

Graficando dichos valores, se obtienen los gráficos de $train_loss$ vs $test_loss$ en la Fig. 34; y de $train_acc$ vs $test_acc$ en la Fig. 35.

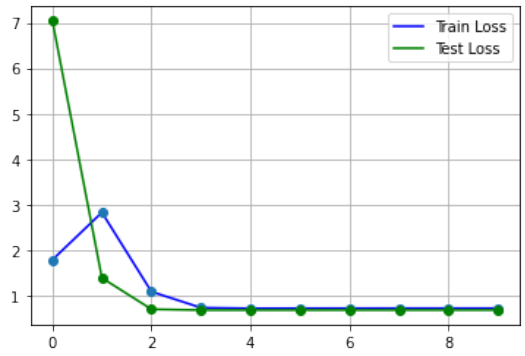

Figura 45: *train_loss* vs *test_loss*, para Adam, shuffle= False

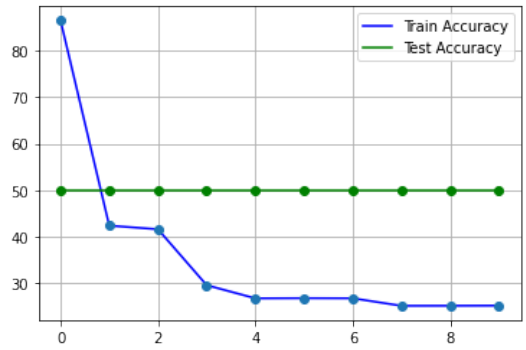

Figura 46: *train_acc* vs *test_acc*, para Adam, shuffle= False

5 Discusión de resultados

5.1 SGD, shuffle = True

En la Fig. 12, podemos observar que el train loss va disminuyendo según lo esperado, mientras que el test loss, primero creció, y luego disminuyó abruptamente, para posteriormente en la época 8 aumentar nuevamente y concluir cayendo finalmente en la época 9. Asimismo, podemos observar en la Fig. 13, como van aumentando ambas magnitudes, según lo que se espera de ellas hasta entregar valores aceptables de $81,11\,\%$ para el train acc; mientras que para el test acc, se obtuvo el valor de $74,15\,\%$.

5.2 SGD, shuffle = False

En este caso, se pueden apreciar los valores de test loss vs train loss en la Fig. 23, donde se aprecia que el train loss se mantiene en valores que tienden a ser cero en todas las épocas, sin embargo el test loss presenta altibajos al igual que en la Fig. 12; sin embargo el cambio es más relevante en la Fig. 24, ya que vemos que para el entrenamiento (train acc), la precisión que ha adquirido roza el $100\,\%$, sin embargo para el test acc, esta no sube del $50\ \%$, esto se debe a que el shuffle al estar en True, almacena el aprendizaje adquirido de cada época y lo va almacenando hasta completar el proceso, sin embargo para shuffle = False, esto no ocurre, y lo que la red aprende en una época es olvidado en la siguiente época.

5.3 Adam, shuffle = True

Podemos observar en la Fig. 34 que el train loss cae directamente, como ocurrió con SGD, shuffle = True, pero podemos ver que el test loss va decreciendo pero manifestando un comportamiento por lo menos oscilatorio, ello puede deberse al learning rate que se estableció, que en este caso

toma el valor de 0.01; dado que adam funciona mejor con *lr* de menor magnitud, puede que esta sea una de la razones; sin embargo, no se disminuyó más el *lr* por el tiempo de ejecución del código. Por otro lado podemos ver en la Fig. 35 que el trin acc crece hasta valores cercanos al 100 %, mientras que el test acc también lo hace pero manifestando un comportamiento oscilatorio.

5.4 Adam, shuffle = False

Los resultados obtenidos en este caso para el train y test loss se puede ver en la Fig. 45, donde se observa un rápido decaimiento hacia el cero para ambos casos, sin embargo en la Fig. 46, podemos ver que el trin acc sufre un descenso hacia el 25 %, ello no debería ocurrir, pues el objetivo es alcanzar el 100 %, asimismo se observa que para el test acc, el valor no cambia del 50 %, cosa que se manifiesta también en la Fig. 24, ello confirma que al colocar shuffle = False, la red neuronal no realiza de manera correcta sus labores, pues olvida el entrenamiento recibido de un época a la siguiente.

5.5 Algunas puntos adicionales

El valor del learning rate debe ser encontrado mediante un tanteo para obtener el mejor resultado, en el caso del optimizador SGD, se recomienda que el valor esté enrte 0 y 1, mientras que para el optimizador Adam, se recomiendan valores de 0.01 o menos; estos valores nos indican qué tanto se desplazará el agoritmo al busca minimizar la función loss, y dar con el valor incorrecto nos podría hacer que oscilemos respecto a un punto (mínimo local) sin llegar al verdadero mínimo del loss; un learning rate muy pequeño también demandará un mayor tiempo de ejecución. Por otro lado, se puede intuir que a un mayor número de época de entrenamiento, se pueden obtener mejores resultados, en este caso solo se emplearon diez épocas, pero podrían ser muchas más.

De lo visto también podemos resaltar la importancia de shuffle = True, para el entrenamiento de las redes neuronales, ya que de esta manera, el aprendizaje adquirido se va almacenando y aplicando en futuras iteraciones del algoritmo mismo, permitiendo así obtener mejores resultados. Finalmente, también es importante la cantidad de datos que se usarán para realizar el entrenamiento, es obvio que con *datasets* más extensos, se obtendrá un mejor aprendizaje y por lo tanto mejore resultados.

6 Conclusiones

Pudimos ver cuatro maneras distintas de trabajar en la clasificación de imágenes, siendo las más eficaces aquellas que tenían shuffle = True, obteniendo precisiones en el testeo de 74,15 % para el optimizador SGD, mientras que para el optimizador Adam se obtuvo una precisión en el teste ode 84,55 %. Para el caso de colocar shuffle = False, se obtuvo que con ambos optimizadores la precisión alcanzada fue del 50 %.

Agradecimientos

Este proyecto fue realizado gracias al apoyo del grupo de AEPIF, quienes organizaron este *summer school* del cual he podido formar parte. Si bien es cierto, no sabía de la existencia de muchas de las librerías usadas en la elaboración de este proyecto, fue gracias a ellos y sobretodo a las clases dictadas por el instructor *Elvin Muñoz Vega* que pude dar un vistazo a este campo del *deep learning* que me ha sorprendido gratamente de poder ser partícipe de todo lo que podemos realizar con nuestros ordenadores. Reitero mis agradecimientos, y enhorabuena por este tipo de iniciativas; espero que vengan muchas más!!!.

Referencias

[1] Documentación de nn.Linear.

[2] Documentación de nn.Flatten.

[3] Documentación de nn.BatchNorm2d.

[4] Documentación de nn.MaxPool2d.

[5] Clases de Deep Learning del Summer School de AEPIF.