UNIVERSIDAD DIEGO PORTALES

FACULTAD DE INGENIERÍA Y CIENCIAS

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

Inteligencia Artificial

Tarea 4

Profesor:
Victor Reyes

Estudiantes:
Felipe Ulloa
Alex Parada

Santiago, Chile 6 de Diciembre del 2022

Índice

1. Introducción

En el siguiente informe se detalla el proceso que se realiza para poder entrenar una red neuronal ANN, más detalladamente con redes convolucionales, tal red neuronal se implementara con el fin de predecir y clasificar una foto de flor de iris.

Antes que todo hay que explicar lo correspondiente a una red neuronal convolucional. Dicha red corresponde a un tipo de neurona artificial donde las neuronas artificiales corresponden a campos receptivos de una manera muy similar a las neuronas en la corteza visual primaria (V1) de un cerebro biológico. Este tipo de red es una variación de un perceptron multicapa, sin embargo, debido a que su aplicación es realizada en matrices bidimensionales, son muy efectivas para tareas de visión artificial, como en la clasificación y segmentación de imágenes, entre otras aplicaciones.

La idea detrás de las CNN es que se procesan (literalmente) volúmenes de datos. El procesamiento se plasma en los pesos, al igual que en otras ANN. Dichos pesos representan importancia de ciertos objetos o aspectos de la entrada (generalmente imágenes). Asimismo, cada neurona responde únicamente a estímulos provenientes de una región limitada.

Esta red además esta subdividida en capas, las cuales las más representativas son las siguientes.5

- Convolution Layer: Este tipo de capas son las que principalmente extraen las características a partir de los datos que están procesando (un ejemplo pueden ser bordes, colores, orientaciones, etc.). Utilizan un Kernel/filtro, tal Kernel trabaja como una ventana que se desliza sobre la data siendo procesada y aplica una multiplicación con la región observada.
- Pooling Layer :Las capas de Pooling derechamente tienen por objetivo reducir la dimensionalidad de los datos. Trabajan también con un tamaño de ventana. Este elemento es clave en el ahorro del uso de recursos de procesamiento.
- Fully Connected Layer:En una capa completamente conectada, cada una de las salidas en la capa previa está conectada con un peso a cada una de las neuronas en la capa. El número de pesos en este tipo de capa incrementada rápidamente cuando el número de entrada o el número de neuronas en la capa incrementa.

Para finalizar, se realiza un análisis de los resultados obtenidos a partir de clasificar una foto de flor de iris y predecir el parámetro de cualquier foto del dataset, dicho análisis se hace en torno a los parámetros de entrada que se le entregan a la red neuronal, exponiendo métricas de error que avalen la red y su funcionamiento, y finalmente comentar la evolución de la red.

2. Metodología

2.1. Actividades previas

2.1.1. Librerías

Para comenzar la tarea, se realiza la importación de las siguientes librerías que nos permite realizar la red neuronal solicitada. Destacar numpy y panda para la lectura de los datos, Torch[5] para crear la Red Neuronal, y maplotlib para realizar los gráficos resultados.

```
#!pip install torchvision numpy pandas torch matplotlib

import matplotlib.pyplot as plt
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import torch
import torchvision
from torch.utils.data import random_split
from torchvision.datasets import ImageFolder
from torchvision import transforms
from torchvision.transforms import ToTensor
from torch.utils.data.dataloader import DataLoader
import torch.nn as nn
import torch.nn.functional as F
```

Figura 1: Librerías utilizadas

2.1.2. Lectura y estandarización de las imágenes.

Se importa la carpeta, desde Drive, que contienen las imágenes junto con su etiqueta, en donde la etiqueta es el nombre de las 3 carpetas que contienen las fotos de su respectiva categoría.

Luego, con el método *transforms.Compose()* de *torchVision* se tranzó las imágenes a valores númericos en una matriz de 2 dimensiones, para luego normalizar las matrices quedando del mismo tamaño *224 x 224*. La normalización se realiza para que todas las imágenes tengan el mismo peso al entrenar el modelo.

Comentar que se tienen 3 grupos de flores, Setosa 67 imágenes, Versicolour 268 imágenes y Virginica 84 imágenes.

```
transformer = torchvision.transforms.Compose(

    [  # Applying Augmentation
        torchvision.transforms.Resize((224, 224)),
        torchvision.transforms.RandomHorizontalFlip(p=0.5),
        torchvision.transforms.RandomVerticalFlip(p=0.5),
        torchvision.transforms.RandomRotation(30),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(

            mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
        ),
    ]

)
dataset = ImageFolder("/content/drive/MyDrive/IA/DataSetT4/Iris", transform=transformer)
```

Figura 2: Lectura y Normalización de Imágenes.

2.1.3. Separación de datos

Ya teniendo las imágenes como matrices *224 x 224* con valores numéricos, se mencionarán como datos.

Para poder entrenar la red, se debe en primera instancia separar el dataset en conjuntos disjuntos de los cuales se calificaran en Train(80 %), Validation(10 %) y Test(10 %). Finalmente se empaquetan los grupos de datos en Dataloaders, con un *Batchsize=32*

```
numDatos = len(dataset)
validation_size = int(numDatos*0.2)
training_size = numDatos - validation_size
train_ds, val_ds_main = random_split(dataset,[training_size, validation_size])
val_ds, test_ds  = random_split(val_ds_main,[int(validation_size/2), int(validation_size/2)])
print(len(train_ds), len(val_ds), len(test_ds))# 337, 42, 42

train_dl = DataLoader(train_ds, batch_size = 32, shuffle=True)
val_dl = DataLoader(val_ds, batch_size = 32)
test_dl = DataLoader(test_ds, batch_size = 32)

print(len(train_dl), len(val_dl), len(test_dl))# 8, 3, 3

#Transformar los DataLoader a Tensor CUDA
train_dl = DeviceDataLoader(train_dl, device)
val_dl = DeviceDataLoader(val_dl, device)
test_dl = DeviceDataLoader(test_dl, device)
```

Figura 3: Separación y empaque de datos.

2.2. Clasificación de flores

2.2.1. Modelo a entrenar para clasificación

Para el apartado de clasificación se procede a diseñar una red convolución divida por capas las cuales son aplicadas por medio de funciones de Torch.

El orden de aplicar las capas en los datos de entrada esta definido por el método *nn.Sequential()*, en donde se puede apreciar 2 repeticiones de un grupo de funciones:

1. **Conv2d**(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)
2. **ReLU**()
3. **Conv2d**(in_channels=16, out_channels=16, kernel_size=5, stride=1, padding=2)
4. **ReLU**()
5. **MaxPool2d**(2, 2)

La primera función a aplicar es Conv2d, la cual es una capa de convolución en donde a los datos de entrada se les multiplica por un kernel de tamaño 3, y separado entre multiplicaciones por un 1 y finalmente agregando un padding de 1 la de esta capa.

Luego se aplica la función de activación ReLU, la que consiste en pasar el valor recibido a la siguiente capa, si es que este es mayor a 0.

Siguiendo con la secuencia, se aplica la capa Conv2d, pero ahora con un tamaño de kernel de 5, un salto entre multiplicación de 1 y un padding de 2.

Otra vez la función de activación ReLU.

Finalmente se aplica una capa Pooling layer, con la técnica de Maximizaron. Esta técnica consiste en dado una ventana, por ejemplo de tamaño 2, elegir el mayor valor entre los abarcados en una ventana y que este valor represente en una nueva matriz resumida el valor del grupo abarcado por la ventana. Por lo que en esta capa se reducen el tamaño de las fotos de 224 a 112.

Luego de estas 2 repeticiones de grupo de funciones, viene una convolución y Relu igual al 1 y 2 del grupo, pero luego la segunda convolución cambia, aparte del tamaño del Kernel, en un Stride 2 y Padding 3, en donde utilizando la fórmula que sale en la Documentación [5], se puede calcular que el tamaño de la imagen se reduce a la mitad.

Para concluir se aplica la función *Flatten()* que consiste en *"Quitar"* una dimensión de la salida, y juntar los elementos como si fuera un solo canal.

Al tener solo un canal, se va reduciendo los valores con el método *Linear()* y limpiando valores con la función de actividad *ReLU()* hasta finalmente reducir los valores que representa a una imagen en solo 4 parámetros.

```python
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

class ImageClassificationModel(nn.Module):
    def training_step(self, batch):
        images, labels = batch

        out = self(images)                  # Generate predictions
        print("Output entrenamiento", out)
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss

    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(

        nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1),
        nn.ReLU(),
        nn.Conv2d(in_channels=16, out_channels=16, kernel_size=5, stride=1, padding=2),
        nn.ReLU(),
        nn.MaxPool2d(2, 2), #output 16 x (224/2) x 112

        nn.Conv2d(in_channels=16, out_channels=64, kernel_size=3, stride=1, padding=1),
        nn.ReLU(),
        nn.Conv2d(in_channels=64, out_channels=64, kernel_size=5, stride=1, padding=2),
        nn.ReLU(),
        nn.MaxPool2d(2, 2), #output 64 x (112/2) x 56

        nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1),
        nn.ReLU(),
        nn.Conv2d(in_channels=128, out_channels=128, kernel_size=7, stride=2, padding=3),
        #  output:  [ 56-1+(2x3)-[(1)x (7-1)] / 2 ] + 1 = 28
        nn.ReLU(),

        nn.Flatten(),
        nn.Linear(128*28*28, 1024),
        nn.ReLU(),
        nn.Linear(1024, 512),
        nn.ReLU(),
        nn.Linear(512, 4)
        )

    def forward(self, xb):
        return self.network(xb)
```

Figura 4: Clasificacion de flores.

El porqué se tienen estos valores específicos como parámetros, por ejemplo Conv2d con tamaño de kernel 3 y salto 1, se debe a 2 razones, la primera es una razón empírica de haber probado distintas combinaciones de secuencias e hiperparametros y la segunda es por suposición a partir de lo aprendido en clases.

Yendo más en concreto se especificó en la primera convolución un parámetro menor de Kernel, dado que la idea es ir *afectando* progresivamente a la imagen, para que los valores no se alejen de la imagen inicial. A partir de estas convoluciones se espera que algunos valore resultantes sean menor a 0, por lo que se procede a *Limpiar* estos valores con la función de activación **ReLU()** y se espera que varios valores se limpien, por lo que ahora se aplica una convolución con Kernel más agresivo (mas grande), y de la misma manera limpiando los valores con **ReLU()** y finalmente dando que se ha aumentado la cantidad de canales, la idea es reducir por otro lado, en este caso por el tamaño de las imágenes con textbfMaxPool2d(2, 2). La razón del porqué reducir con Max y no con Average, es que la idea de la clasificación es que clasifique a partir de las características de la flor que son características dominantes en la foto, y no por un conjunto de características de la foto que abarcan por ejemplo características del ambiente que de la misma flor.

La tercera parte de la secuencia busca simular el patrón anterior, de tener una convolución suave, limpiar, convolución agresiva y reducir. Sin embargo las últimas 2, de convolución agresiva y reducir se realizan con la misma función **Conv2d()**, dado que tiene un tamaño de kernel más grande y la parte de reducir, se realiza dado que se tiene un stride de 2, lo cual para poder llegar a la mitad de la entrada además se agrega un Padding de 3. Se decidió que esta tercera parte sea así, para poder agregar ruido a través del padding para evitar un Overfitting con las imágenes de entrenamiento.

La parte final consiste en reducir los valores de las imágenes a solo 4 parámetros, que representaran las 4 longitudes que tiene una Flor.

Para finalizar este apartado, se procede a trabajar con el valor de precisión y de perdida que representan nuestras métricas de error, dicha métrica nos permitirá efectuar un análisis frente al cambio que se puede aplicar a los parámetros iniciales y como estos afectan a la red.

```
def validation_step(self, batch):
      images, labels = batch
      out = self(images)                    # Generate predictions
      loss = F.cross_entropy(out, labels)   # Calculate loss
      acc = accuracy(out, labels)           # Calculate accuracy
      return {'val_loss': loss.detach(), 'val_acc': acc}

  def validation_epoch_end(self, outputs):
      batch_losses = [x['val_loss'] for x in outputs]
      epoch_loss = torch.stack(batch_losses).mean()   # Combine losses
      batch_accs = [x['val_acc'] for x in outputs]
      epoch_acc = torch.stack(batch_accs).mean()      # Combine accuracies
      return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

  def epoch_end(self, epoch, result):
      print("Epoch [{}], train_loss: {:.4f}, val_loss: {:.4f}, val_acc: {:.4f}".format(


          epoch, result['train_loss'], result['val_loss'], result['val_acc']))
```

Figura 5: Funciones de validaciones.

2.2.2. Entrenar el modelo con diferentes ciclos

Se procede a entrenar el modelo con distintas cantidades de periodos, con el fin de evidenciar cuál es el valor óptimo de Periodos de entrenamientos, para poder clasificar una imagen. Lo anterior también se realiza a partir de un mismo valor de precisión y función de optimización representada por los siguientes valores.

```
num_epochs = 50
opt_func = torch.optim.Adam
lr = 0.001
```

Figura 6: Valores de entrenamiento.

Los ciclos de entrenamientos se basan en cuantas veces se ejecutan las imágenes en el modelo esto se realiza a partir de los parámetros iniciales, los Epoch y los valores correspondientes a la precisión y optimización. Luego para comprobar los valores obtenidos se realiza una fase de validación, con el fin de aplicar la métrica de entrenamiento perdido y poder analizar tal métrica más adelante.

```
 def evaluate(model, val_loader):
    model.eval()
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)


def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        # Training Phase
        model.train()
        train_losses = []
        for batch in train_loader:
            loss = model.training_step(batch)
            train_losses.append(loss)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation phase
        result = evaluate(model, val_loader)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        model.epoch_end(epoch, result)
        history.append(result)
    return history


model = ImageClassificationModel()
history = fit(num_epochs, lr, model, train_dl, val_dl, opt_func)
```

Figura 7: Periodos de entrenamiento.

2.3. Obtención de parámetros de las fotos

2.3.1. Modelo a entrenar para obtención de parámetros

Para el apartado de obtención de parámetros de una foto, se tiene que el modelo es similar al de clasificación, tiene la misma secuencia de aprendizaje y métodos a excepción de **training_step()** que se puede ver en la figura[4]. Sin embargo el enfoque que se aplica para la obtención de parámetros de una foto, es asociar los 4 valores de salida del entrenamiento, antes de clasificación, y relacionarla con su respectiva foto de entrada. Para esto se trabaja con la variable out que permitirá asociar los parámetros con las fotos correspondientes.

Luego se aplica las funciones para trabajar con las métricas de error que permitirán fluctuar un análisis del comportamiento de las variables, estas se irán imprimiendo por cada Epoch con la finalidad de ir viendo los resultados en cada etapa.

```
class ImageClassificationModel(nn.Module):
    def training_step(self, batch):
        images, labels = batch

        out = self(images)                  # Generate predictions
        print("Parametros de la imagen: ", out)
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss
```

Figura 8: Re-escritura del método *trainingstep*().

2.3.2. Entrenar nuevo modelo con diferentes parámetros

Se procede a entrenar el nuevo modelo con una distintas cantidades de periodos, esto se realiza con el fin de evidenciar cuál es el valor óptimo de ciclos con el cual poder predecir los parámetros de las fotos. Además se utiliza el valor de precisión y optimización para poder predecir lo anterior, tales valores están representados por los siguientes valores.

```
num_epochs =20
opt_func = torch.optim.Adam
lr = 0.001
```

Figura 9: Valores de entrenamiento.

Los ciclos de entrenamientos se basan en cuantas veces se ejecutan las imágenes en el modelo esto se realiza a partir de los parámetros iniciales, los Epoch y los valores correspondientes a la precisión y optimización. Luego para comprobar los valores obtenidos se realiza una fase de validación, con el fin de aplicar la métrica de entrenamiento perdido y poder analizar tal métrica más adelante.

Lo anterior es de la misma manera que en el apartado anterior por lo cual, ver la imagen [7]

3. Resultados y Análisis

3.1. Resultados Clasificación de la Red

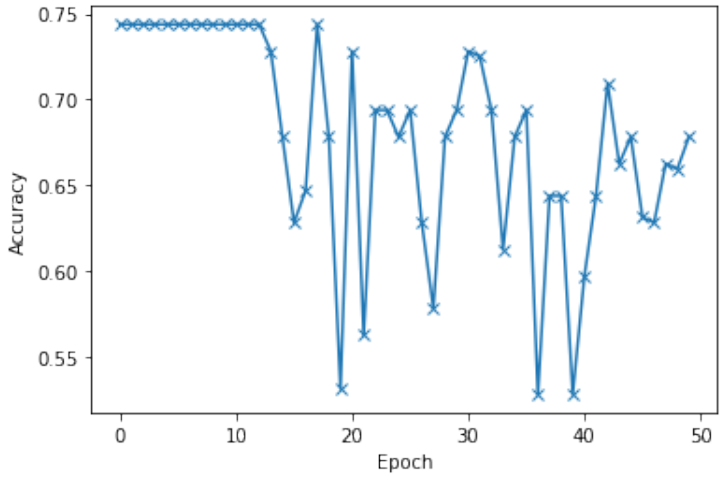

Figura 10: Resultados clasificación de imágenes de Flor.

Para la predicción de parámetros, se obtuvo las correspondientes métricas de error para el último epoch:

1. **Train Loss**(0.7189)
2. **Value Accuracy**(0.6781)
3. **Validation Loss**(0.8678)

Estas métricas en sí no fueron fijas para cada iteración que se abarcó. Por una parte en la métrica de precisión se obtuvo un rango de variación entre 0.52 a 0.74 en cada Epoch iterado, sin embargo el peak no fue obtenido en la última iteración, esto surge a que los Epoch pueden aumentar el valor de precisión hasta cierto límite , al sobrepasar este limite el modelo tiende a sobre ajustarse lo que ocasiona que se haya adaptado tanto a los valores anteriores de entrada, que al predecir otra imagen, no logre acertar, por ende el peak de precisión surge luego de los primeros epoch y antes del primer decrecimiento, tal como se aprecia en la figura [13].

Por otra parte, se evidencia que la métrica train loss yace bajo el rango de 0.719 a 1.65, esto quiere decir que el modelo se está adaptando fácilmente a clasificar las imágenes, ya que esta métrica busca como un modelo de aprendizaje se ajusta a los datos de entrenamiento, sin embargo esta métrica se debe comparar con la métrica validation loss, la cual permite validar si el modelo posee un sobre ajuste, esto último se ocasiona cuando la métrica train loss es menor al validation loss. Si se revisa el rango de resultados por cada Epoch en validation loss, se observa que ocurre este suceso, lo que implica que el modelo entrenado posee un sobre ajuste de medidas.

3.2. Resultados de Predicción de la red

Se observa para este apartado que los valores de las métricas poseen un rango similar a los obtenidos en clasificación, sin embargo en las primeras iteraciones se observa que la métrica de train loss esta muy por encima del valor de validation loss, esto ocurre debido a factores como la regularización, ya que esta técnica se utiliza cuando se entrena el conjunto de entrenamiento inflando la métrica del trainloss y por otra parte también, se encuentra el parámetro de dropout (o abandono), dicho parámetro penaliza la varianza que realiza el modelo al congelar aleatoriamente las neuronas en una capa durante el entrenamiento del modelo, como se menciona dicho parámetro solo se aplica cuando se realiza el entrenamiento del modelo y por ende afecta solamente al train loss implicando que esté en tal punto posea un valor mayor al validation loss.5

Al ir aumentando los Epoch el modelo se va ajustando a los datos y se va regularizando los parámetros, como es el caso de la precisión hasta cierto punto (ya que después se va sobre ajustando tal métrica) y por otra parte se va disminuyendo el valor de train loss a tal punto de ser menor a la métrica de validation loss.

Finalmente, se observa en las imágenes posteriores que los valores obtenidos en el output son equivalentes a los parámetros de las fotos del dataset.

```
        grad_fn=<AddmmBackward0>)
Epoch [5], train_loss: 0.9338, val_loss: 0.8160, val_acc: 0.7437
Output entrenamiento tensor([[ 0.4491,  1.7617,  0.9136, -5.4693],
        [ 0.4165,  1.6437,  0.8537, -5.1020],
        [ 0.4178,  1.6442,  0.8541, -5.1051],
        [ 0.3048,  1.2391,  0.6437, -3.8318],
        [ 0.2239,  0.8887,  0.4656, -2.7588],
```

Figura 11: Resultado output epoch 5

```
        grad_fn=<AddmmBackward0>)
Epoch [19], train_loss: 0.8123, val_loss: 0.9191, val_acc: 0.6281
```

Figura 12: Resultado output epoch 19

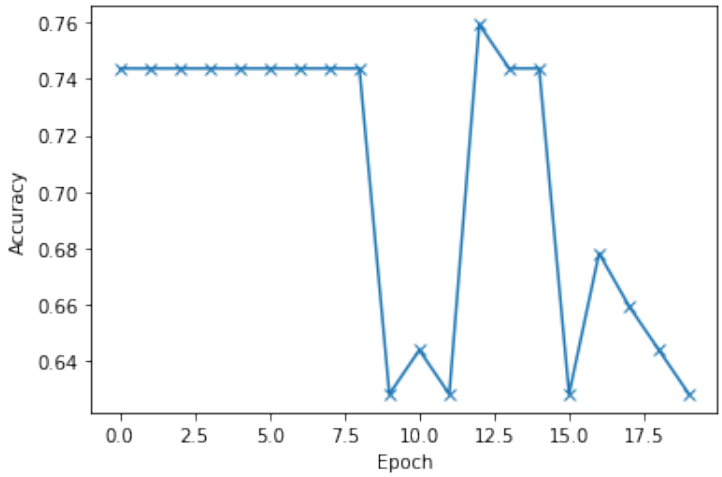

Figura 13: Resultado entrenamiento de obtención de parámetros de una Foto

4. Link Colab

Se adjunta el link del colab, que permite ver directamente el codigo realizado

`https://colab.research.google.com/drive/1aNJN6yo3T7oegr5BAJLFmw2wGZHnRT5T?usp=sharing`

5. Anexo

https://pytorch.org/

https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

https://towardsdatascience.com/what-your-validation-loss-is-lower-than-your-training-loss-this-is-why-5e9

https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenad

https://datascientest.com/es/convolutional-neural-network-es