Universidad Autónoma Chapingo

DIVISION DE CIENCIAS FORESTALES

LICENCIATURA EN ESTADISTICA

INTRODUCCIÓN A LA OPTIMIZACIÓN MATEMÁTICA

ALUMNA: VÁSQUEZ GONZÁLEZ ARELY YAZMIN

Grado: 6º **Grupo:** 5

INTRODUCCIÓN

La inteligencia artificial es una nueva herramienta que ha cobrado gran importancia en los últimos años, una de sus ramas es el Machine Learning el cual crea sistemas que aprenden automáticamente, y que puede ser supervisado o no supervisado; uno de sus subcampos es el aprendizaje profundo que a su vez trata del uso de redes neuronales.

No hay que perder de vista que la estadística es la base fundamental del aprendizaje automático, gracias a ella se pueden analizar los grandes cantidades de datos para así deducir cual es el resultado óptimo, como un ejemplo inmediato podemos pensar en el función de perdida.

Las redes neuronales artificiales están formadas por elementos llamados neuronas que están conectados a otras neuronas. Una neurona artificial es una entidad que recibe unos datos de entrada, les aplica una serie de operaciones matemáticas y un función de activación y genera un resultado; se usa principalmente en problemas de clasificación y predicción. El futuro de las redes neuronales es muy prometedor, gracias a ella se han hecho importantes avances en la ciencia, como un ejemplo podemos mencionar el reciente éxito de AlphaFold 2 en la predicción del plegamiento de proteínas, ofreciendo así una herramienta que no solo es más barata y rápida que los métodos que se venían utilizando (cristalografía de rayos X o la resonancia magnética nuclear) sino que tiene un gran precisión, y lo más importante aún es que esto podría ayudar en el desarrollo de nuevos fármacos para tratar enfermedades.

En este proyecto se trabajará con un problema de clasificación específicamente en el reconocimiento de imágenes, y que mejor opción que usar una red neuronal convolucional. La redes de visión por computadora han ido adquirido un mayor peso a diferencia de otro tipo de redes debido a su uso en diversos campos, tal como en los sectores de electrónica, seguridad, automotriz, alimentación, medica, etc.

Objetivo generales:

Ser capaz de usar los conocimientos adquiridos en el curso de IOM, para implementar una red neuronal.

Objetivos específicos:

Clasificar una base de datos de imágenes de rayos X de tórax en tres categorías, casos positivos de COVID-19, opacidad pulmonar (infección pulmonar no COVID) y normal.

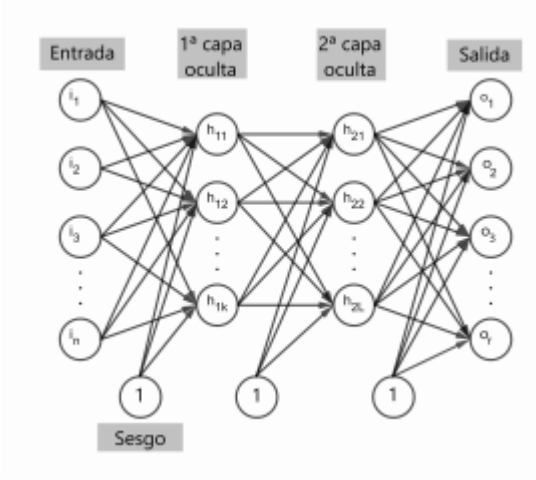
Motivación

La pandemia causada por la nueva enfermedad del Coronavirus (COVID-19) ha afectado en gran medida el mundo, además ha cambiado nuestro formas de vida y definido una nueva, por eso mismo el análisis de datos generados es de gran importancia para ver el seguimiento de esta enfermedad, hacer predicciones, y entender un poco más sobre ella.

El diagnostico temprano de esta enfermedad puede ayudar a minimizar los contagios, y a obtener un resultado exitoso en el tratamiento. Se utilizan varias técnicas en el diagnóstico, una de ellas son las imágenes de radiografía de tórax, que tiene como ventaja la capacidad de realizarlas fácilmente utilizando equipos de rayos X portátiles. Utilizar modelos de aprendizaje profundo, pueden ayudar a detectar COVID-19 con más rapidez y una buena precisión si es que se cuenta con un buen modelo, hay que añadir que además de detectar COVID-19, se puede crear una red neuronal que también pueda identificar mas enfermedades relacionadas como la neumonía o infección pulmonar.

Un poco de teoría

Una red neuronal se compone de neuronas y las conexiones entre ellas, en donde los valores de la señal de entrada se multiplican por pesos, luego se suman los productos y después de utiliza una función de activación para generar la salida de la neurona que puede ser enviada a otras neuronas de la red.



En general en una red neuronal se encuentran tres tipos de capas, una capa de entrada, una o más capas ocultas y una capa de salida, esta última es la que se encarga de mostrar los resultados ya sean de predicción o clasificación.

El proceso de entrenamiento consiste en encontrar el valor óptimo de los parámetros de la red, el algoritmo para lograr esto es el de Back-propagation, que constan de dos

partes el paso hacia adelante y hacia atrás; para la optimización de los pesos se utiliza el descenso del gradiente.

Se cuentan con varios funciones de activación a escoger:

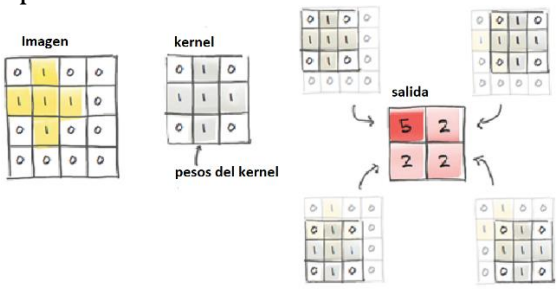
función ReLU: anula los valores negativos y deja los positivos igual a como entran.

Función lineal: es parecida a la ReLU, pero aquí no se anulan los valores negativos.

Función sigmoide: está en un rango de salida entre cero y uno, generalmente interpretada como una probabilidad.

Función softmax: muy utilizada para los problemas de clasificación.

Las redes neuronales convolucionales generalmente tendrán tres tipos de entrada, una capa convolucional, una capa de reducción generalmente Max-Pooling, y una capa completamente conectada o lineal, que nos dará la salida.
Las CNN tienen varias capas ocultas, pero además estas tienen una jerarquía, en las primeras capas de convolución puede aprender patrones locales como los bordes o líneas, en la segunda capa aprenderá patrones más grandes hechos de las primeras capas, y así sucesivamente se van especializando hasta reconocer y aprender formas más complejas y abstractas, otra característica de estas es que el aprendizaje de los patrones es invariante a la traslación.
Convoluciones: La operación de convolución aplica sobre la imagen un filtro o kernel el cual devuelve un mapa de las características, esta operación reduce el tamaño original de la imagen, que también depende del tamaño del kernel. Los filtros codifican aspectos específicos de los datos de entrada.
En PyTorch se definen por tres parámetros claves: el número de canales, numero de salidas y el tamaño del kernel.
Es importante mencionar que las convoluciones operan sobre tensores 3D, alto ancho y eje de canales, las imágenes a color tienen tres ejes de canales, por lo que el filtro será de 3 planos, que se sumaran para así obtener una salida.
Con el kernel se hace un producto escalar con un grupo de pixeles de la imagen, y después el kernel recorrerá todas las neuronas de entrada obteniendo una matriz de salida, que será la nueva capa oculta. En la siguiente imagen se puede visualizar como opera lo descrito anteriormente.



Las dimensiones espaciales de la entrada cambian, pero si se desea obtener el mismo tamaño se puede usar padding, el cual agrega pixeles de valor cero alrededor de la imagen original.

Max-Pooling. Una de sus funciones es la reducción de las dimensiones espacial y consiste en extraer ventanas de los mapas de características y quedarse con el máximo valor de cada canal. Generalmente se realiza con ventanas de 2x2 y zancada igual a 2, esto reduce el tamaño de la imagen en 2, mientras que en la convolución generalmente la zancada es 1 y el kernel es de 3x3 o de 5x5.

DESARROLLO

1.- Obtención de datos.

2. Preparación de datos.

3. Creación del modelo

4. Entrenamiento del modelo

5. Evaluación del modelo

Obtención de datos

La base de datos de imágenes se descargó de la plataforma *Kaggle,* fue creado por investigadores de la Universidad de Qatar, Doha, y de la universidad de Dhaka, Bangladesh. La base de datos son imágenes de rayos X de tórax para casos positivos de COVID-19 junto con imágenes de infección pulmonar (no COVID-19) e imágenes normales.

La base de datos con la que se trabajó tiene tres carpetas: entrenamiento, validación y prueba, en la carpeta de entrenamiento se tienen 7500 imágenes 2500 para cada caso, para validación se tienen 1950 imágenes 650 para cada caso y para los datos de prueba se tienen 1350 imágenes.

En primer lugar, se importan todas los paquetes necesarios para crear nuestro red neuronal.

```
import os
import numpy as np
import matplotlib.pyplot as plt
import shutil
from torchvision import transforms
from torchvision import models
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import lr_scheduler
from torch import optim
from torchvision.datasets import ImageFolder
from torchvision.utils import make_grid
from torch.utils.data import Dataset, DataLoader
%matplotlib inline
import torchvision
import torchvision.transforms as transforms
```

Preparación de datos.

Cargaremos el conjunto de datos de las imágenes, especificando la dirección de la carpeta donde están ubicadas.

```
carpeta_entrenamiento = "C:/Users/yaya9/Documents/RADIOGRAFÍA COVID-19/entrenamiento"
carpeta_validacion = "C:/Users/yaya9/Documents/RADIOGRAFÍA COVID-19/validacion"
carpeta_prueba ="C:/Users/yaya9/Documents/RADIOGRAFÍA COVID-19/prueba"
```

Para el tratamiento de las imágenes se utilizó transfroms. Compose.

Transfroms. Compose permite hacer varias transformaciones juntas, y se aplican a todas las entradas una por una, algunas de las transformaciones que se pueden usar son:

transforms.RandomResizedCrop(): extrae un parche del tamaño indicado de la imagen de entrada al azar.

transforms.ToTensor(): Convierte la imagen de entrada a tensor PyTorch.

transforms.Normalize(): Normaliza una imagen tensorial.

transforms.Resize():cambia el tamaño de las imágenes de entrada

transforms.CentreCrop(): Recorta la parte central de la imagen.

transforms.RandomHorizontalFlip(): Voltea horizontalmente la imagen dada aleatoriamiente.

torchvision.transforms.RandomCrop (): recortara la imagen dada en ubicaciones aleatorias para crear imágenes para el entrenamiento.

torchvision.transforms.Grayscale (): convierte la imagen a escala de grises.

Para las imágenes usadas solo se usaron tres de estas transformaciones, las imágenes serán redimensionadas a 200x200 pixeles, también se convirtieron las imágenes a tensores Pytorch, y se normalizaron las imágenes tensoriales, la primera tupla es la media y la segunda es la desviación estándar, cada una de estas tuplas tiene tres parámetros porque se tiene tres canales. Los valores que se están usando para la normalización son tomados de *imagenet.*

```
transformacion= transforms.Compose([transforms.Resize((200,200))
                                    ,transforms.ToTensor()
                                    ,transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
                                   ])
```

Creación de dataset y dataloader

Torch.utils.Dataset y torch.utils.Dataloader, nos facilita el trabajo para procesar los datos. Dataset almacena las muestras y sus etiquetas correspondientes, mientras que DataLoader nos permite crear lotes dadas las transformaciones y el directorio de datos que se hicieron con la clase Transforms e ImageFolder.

```
dataset_entrenamiento = ImageFolder(carpeta_entrenamiento, transformacion)
dataset_validacion = ImageFolder(carpeta_validacion,transformacion)
dataset_prueba = ImageFolder(carpeta_prueba,transformacion)
```

```
datal_entrenamiento = DataLoader(dataset_entrenamiento, batch_size=4, shuffle=True)
datal_validacion = DataLoader(dataset_validacion, batch_size=32, shuffle=True)
datal_prueba = DataLoader(dataset_prueba, batch_size=3, shuffle=True)
```

```
print(dataset_entrenamiento.class_to_idx)
print(dataset_entrenamiento.classes)
```

```
{'COVID -19': 0, 'normal': 1, 'opacidad pulmonar': 2}
['COVID -19', 'normal', 'opacidad pulmonar']
```

La siguiente función nos sirve para visualizar las imágenes, dentro de esta función se remodelan los tensores y desnormalizan los valores.

```
def Graficar(entrada):
    entrada = entrada.numpy().transpose((1, 2, 0))
    media = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    entrada = std * entrada + media
    entrada = np.clip(entrada, 0, 1)
    plt.imshow(entrada)
```

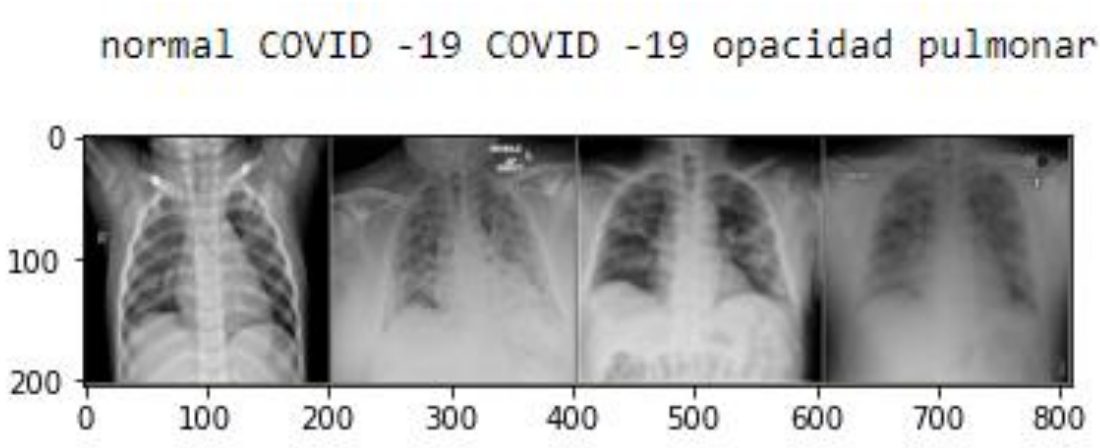
np.clip(): sujeta todos los elementos de la entrada dentro del rango especificado, para asegurarse que este en el rango adecuado.

Veamos una imagen de los datos de entrenamiento usando la función anterior. Las clases a la que pertenecen las imágenes se obtienen usando dataset. Se usará iter para iterar nuestro dataloader (cargador de datos), y next ayudará a iterar sobre él, esto nos permitirá obtener algunas imágenes al azar, dependiendo del tamaño de lote especificado en *dataloader*.

```
clases = datal_entrenamiento.dataset.classes
```

```
dataiter = iter(datal_entrenamiento)
imagenes, etiquetas = dataiter.next()
Graficar(torchvision.utils.make_grid(imagenes))
print('  ', ' '.join('%8s' % clases[etiquetas[j]] for j in range(4)))
```

```
dataiter = iter(datal_entrenamiento)
imagenes, etiquetas = dataiter.next()
Graficar(torchvision.utils.make_grid(imagenes))
print('   ', ' '.join('%8s' % clases[etiquetas[j]] for j in range(4)))
```



torchvision.utils.make_grid() crea un lote de imágenes en una cuadricula.

Creación de la arquitectura

Para crear nuestro modelo antes se debe de crear la arquitectura como una clase Python

Primero se deben de inicializar las capas. Pasar una función de inicialización a nn.Module, inicializará los pesos en la totalidad.

En esta arquitectura se usaron dos capas de convolución, en la primera capa se tiene como entrada 3 ya que se tienen tres mapas de características y se pide como salida 32 mapas de características, la segunda capas de convolución tiene como entrada la salida de la capa de convolución anterior y se generan 64 mapas de características, el kernel usado es de 5x5. También se están usando dos capas lineales para calcular la entrada de la primera capa lineal se hicieron los siguientes cálculos:

Canales	alto	ancho
3	200	200
32	196	196
32	98	98
64	94	94
64	47	47

Entrada de la primera capa lineal = 64 x 47 x 47=141,376.

en la segunda capa lineal la salida es 3, porque se tienen tres categoría (normal, COVID, opacidad pulmonar).

Forward implementa la propagación hacia adelante. Se está usando max_pooling con zancada igual a 2 de 2x2. Dropout (abandono) solo se aplica para entrenamiento ésta pone en cero algunos pesos y al final se aplica la función log_softmax.

```
class Covid(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv_1 = nn.Conv2d(3, 32, kernel_size=5)
        self.conv_2 = nn.Conv2d(32, 64, kernel_size=5)
        self.lineal_1 = nn.Linear(141376, 64)
        self.lineal_2 = nn.Linear(64,3)


    def forward(self, entrada):
        x = F.relu(F.max_pool2d(self.conv_1(entrada), 2))
        x = F.relu(F.max_pool2d((self.conv_2(x)), 2))
        x = x.view(-1,141376)
        x = F.relu(self.lineal_1(x))
        x = self.lineal_2(x)
        x = F.dropout(x, training=self.training)
        salida = F.log_softmax(x,dim=1)
        return salida
```

Conv2D (): representa la capa de convolución, aplica una convolución 2D sobre una señal de entrada compuesta por varios planos de entrada.

Relu: permite una reducción de la salida de la neurona cuando no esté activa (transforma los valores introducidos anulando los valores negativos y dejando los positivos).

Max_pooling2D (): realiza el proceso de max-pooling con un núcleo de dimensión 2x2.

Dropout(): desactiva las neuronas para evitar el sobreajuste.

Después para crear el modelo instanciamos un objeto con la clase COVID.

```
modelo = Covid()
```

Función de perdida y optimizador

Se eligió el optimizador SGD con una tasa de aprendizaje de 0.01 y momentum 0.5, en SGD, los pesos se actualizan después de recorrer cada muestra de entrenamiento.

Cuando se entrena un modelo al inicio generalmente la perdida disminuye rápidamente pero gradualmente puede llegar a un punto en el que parece no estar progresando, y es aquí donde se hace el uso de momentum, que evita quedarse atascado en un mínimo local.

La función de perdida elegida es CrossEntropyLoss(), se escogió esta función de perdida porque es útil al entrenar un problema de clasificación con C clases.

```
optimizador = optim.SGD(modelo.parameters(), lr=0.01, momentum=0.5)
f_perdida=nn.CrossEntropyLoss()
```

Entrenamiento del modelo

```
def entrenar_modelo(epoca, modelo, cargador_datos, fase='entrenamiento',volatile=False):
    if fase == 'entrenamiento':
        modelo.train()
    if fase == 'validacion':
        modelo.eval()
        volatile=True
    perdida_aux = 0.0
    correcto_aux= 0
    for indice_lote, (datos,etiqueta) in enumerate(cargador_datos):
        datos, etiqueta = Variable(datos,volatile),Variable(etiqueta)
        if fase == 'entrenamiento':
            optimizador.zero_grad()
        salida = modelo(datos)
        perdida = f_perdida(salida,etiqueta)

        perdida_aux += f_perdida(salida,etiqueta).data
        preds = salida.data.max(dim=1,keepdim=True)[1]
        correcto_aux+= preds.eq(etiqueta.data.view_as(preds)).cpu().sum()
        if fase == 'entrenamiento':
            perdida.backward()
            optimizador.step()

    perdida = perdida_aux/len(cargador_datos.dataset)
    precision = 100. * correcto_aux/len(cargador_datos.dataset)

    if fase == "entrenamiento":
        print(f"Epoca: {epoca}")
    print(f"La pérdida de {fase} es: {perdida:{3}.{3}} y la presición de {fase} es {correcto_aux}/{len(cargador_datos.dataset)}
    {precision:{10}.{4}}")
    return perdida, precision
```

Modelo.train() se llama en la fase de entrenamiento y model.eval() se llama en la fase de evaluación.

Primero se inicializan la perdida y precisión a cero. Después al pasar un cargador de datos a la función se obtienen datos de imágenes y etiquetas.

En fase==’entrenamiento’, se pone optimizer.zero_grad () porque PyTorch almacena gradientes en la memoria y por eso mismo se tiene que poner en cero en cada paso después se pasan las entradas a nuestro modelo.

Se calcula la pérdida con la función de perdida especificada anteriormente usando la etiqueta y las salidas.

Después de calcular la pérdida, se calcula el gradiente de los parámetros con perdida.backward() y se actualizan los parámetros usando este gradiente usando optimizador.step().

Posteriormente, se ejecuta la función creada anteriormente para entrenar el modelo para los datos de prueba y validación, durante 22 épocas.

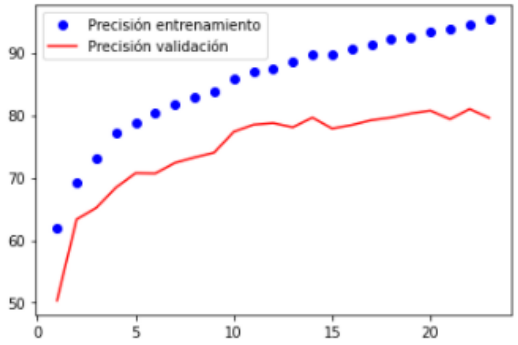
```
perdidas_entrenamiento , precision_entrenamiento = [],[]
perdidas_validacion , precision_validacion = [],[]
for epoca in range(1,23):
    perdida_epoca, precision_epoca = entrenar_modelo(epoca,modelo,dl_entrenamiento,fase='entrenamiento')
    perdida_epoca_validacion , precision_epoca_validacion = entrenar_modelo(epoca,modelo, dl_validacion,fase='validacion')
    perdidas_entrenamiento.append(perdida_epoca)
    precision_entrenamiento.append(precision_epoca)
    perdidas_validacion.append(perdida_epoca_validacion)
    precision_validacion.append(precision_epoca_validacion)
```

Ultimas épocas:

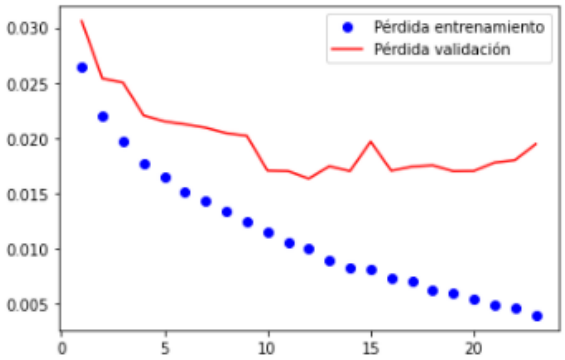
```
La pérdida de entrenamiento es: 0.00895 y la presición de entrenamiento es 6643/7500     88.57
La pérdida de validacion es: 0.0175 y la presición de validacion es 1522/1950     78.05
Epoca: 14
La pérdida de entrenamiento es: 0.00831 y la presición de entrenamiento es 6729/7500     89.72
La pérdida de validacion es: 0.017 y la presición de validacion es 1552/1950     79.59
Epoca: 15
La pérdida de entrenamiento es: 0.0081 y la presición de entrenamiento es 6724/7500     89.65
La pérdida de validacion es: 0.0197 y la presición de validacion es 1518/1950     77.85
Epoca: 16
La pérdida de entrenamiento es: 0.0073 y la presición de entrenamiento es 6796/7500     90.61
La pérdida de validacion es: 0.0171 y la presición de validacion es 1529/1950     78.41
Epoca: 17
La pérdida de entrenamiento es: 0.00708 y la presición de entrenamiento es 6841/7500     91.21
La pérdida de validacion es: 0.0174 y la presición de validacion es 1544/1950     79.18
Epoca: 18
La pérdida de entrenamiento es: 0.00622 y la presición de entrenamiento es 6920/7500     92.27
La pérdida de validacion es: 0.0175 y la presición de validacion es 1552/1950     79.59
Epoca: 19
La pérdida de entrenamiento es: 0.00596 y la presición de entrenamiento es 6937/7500     92.49
La pérdida de validacion es: 0.017 y la presición de validacion es 1565/1950     80.26
Epoca: 20
La pérdida de entrenamiento es: 0.00547 y la presición de entrenamiento es 7002/7500     93.36
La pérdida de validacion es: 0.017 y la presición de validacion es 1574/1950     80.72
Epoca: 21
La pérdida de entrenamiento es: 0.00489 y la presición de entrenamiento es 7027/7500     93.69
La pérdida de validacion es: 0.0178 y la presición de validacion es 1547/1950     79.33
Epoca: 22
La pérdida de entrenamiento es: 0.00458 y la presición de entrenamiento es 7087/7500     94.49
La pérdida de validacion es: 0.018 y la presición de validacion es 1579/1950     80.97
```

Grafica de precisión y pérdida.

```
plt.plot(range(1,len(precision_entrenamiento)+1),precision_entrenamiento,'bo',label = 'Precisión entrenamiento')
plt.plot(range(1,len(precision_validacion)+1),precision_validacion,'r',label = 'Precisión validación')
plt.legend()
```




```
plt.plot(range(1,len(perdidas_entrenamiento)+1),perdidas_entrenamiento,'bo',label = 'Pérdida entrenamiento')
plt.plot(range(1,len(perdidas_validacion)+1),perdidas_validacion,'r',label = 'Pérdida validación')
plt.legend()
```



La precisión alcanzada para entrenamiento fue de 94.49%, y la perdida fue de 0.00458

La precisión para validación fue de 80.97% y la perdida fue de 0.018

Evaluación del modelo

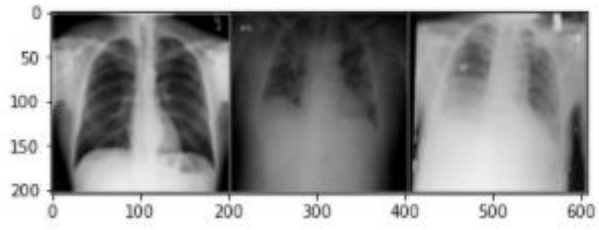
Para la evaluación del modelo se usará nuestro conjunto de datos de prueba.

Veamos que tal predice algunas imágenes.

Primero se visualizarán las imágenes con su respectivas etiquetas reales. Después se vera la predicción que hace nuestro modelo. Se están visualizando tres imágenes porque en el dataloader de prueba se especificó que los lotes sean igual a tres.

```
clases = datal_prueba.dataset.classes
dataiter = iter(datal_prueba)
imagenes, etiquetas = dataiter.next()
Graficar(torchvision.utils.make_grid(imagenes))
print('Categoria real a la que pertenecen las imagenes: ', ' '.join('%5s' % clases[etiquetas[j]] for j in range(3)))
```

```
Categoria real a la que pertenecen las imagenes:  normal COVID -19 opacidad pulmonar
```




```
salidas = modelo(imagenes)
_, prediccion = torch.max(salidas, 1)
print('Prediccion que hace nuestro modelo: ', ' '.join('%5s' % clases[prediccion[j]]
                              for j in range(3)))
```

```
Prediccion que hace nuestro modelo:  normal COVID -19 opacidad pulmonar
```

La precisión de todos los datos de prueba es:

```
correcto = 0
total = 0
with torch.no_grad():
    for data in dl_prueba:
        imagenes, etiquetas = data
        salidas = modelo(imagenes)
        _, prediccion = torch.max(salidas.data, 1)
        total += etiquetas.size(0)
        correcto += (prediccion == etiquetas).sum().item()
print('Precision en los datos de prueba: %d %%' % (100 * correcto / total))
```

```
correcto = 0
total = 0
with torch.no_grad():
    for data in dl_prueba:
        imagenes, etiquetas = data
        salidas = modelo(imagenes)
        _, prediccion = torch.max(salidas.data, 1)
        total += etiquetas.size(0)
        correcto += (prediccion == etiquetas).sum().item()

print('Precision en los datos de prueba: %d %%' % (100 * correcto / total))
```

```
Precision en los datos de prueba: 79 %
```

Precisión para cada clase:

```
clase_correcto = list(0.for i in range(3))
clase_total = list(0.for i in range(3))
with torch.no_grad():
    for data in dl_prueba:
        imagenes, etiquetas = data
        salidas = modelo(imagenes)
        _, prediccion = torch.max(salidas, 1)
        c=(prediccion==etiquetas).squeeze()
        for i in range(3):
            etiqueta=etiquetas[i]
            clase_correcto[etiqueta] += c[i].item()
            clase_total[etiqueta]+=1
for i in range(3):
    print('Precision %4s: %2d %% ' % (clases[i], 100*clase_correcto[i]/clase_total[i]))
```

```
Precision COVID -19: 82 %
Precision normal: 74 %
Precision opacidad pulmonar: 82 %
```

Conclusión

Se logró clasificar las imágenes en tres categorías implementando redes neuronales convolucionales, aunque la precisión no fue la mejor, pero sabemos que nos queda mucho por aprender.

BIBLIOGRAFÍA

https://towardsdatascience.com/improves-cnn-performance-by-applying-data-transformation-bf86b3f4cef4

https://rdrr.io/github/mlverse/torchvision/man/vision_make_grid.html

https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

https://docs.python.org/2/library/functions.html

https://www.kaggle.com/tawsifurrahman/covid19-radiography-database