UNIVERSIDAD NACIONAL DE INGENIERÍA | Facultad de Ingeniería Industrial y de Sistemas

MAESTRÍA EN INTELIGENCIA ARTIFICIAL

Curso de Redes Neuronales y Aprendizaje Profundo

Integrantes

Kevin Gómez Villanueva
Umbert Lewis de la Cruz Rodriguez
Fernando Boza Gutarra
Yovany Romero Ramos

Pregunta 1.- Hacer las operaciones de forward y backward propagation de forma manual para un MLP que tome 3 entradas (3x1), tenga 2 hidden layers de tamaño 4, y una salida de 1x1. El cálculo lo deben hacer para la siguiente data:

La data de entrada X y el vector Y deseado es el siguiente:

$$X_s = \begin{bmatrix} 2.5 & 3.5 & -0.5 \\ 4.0 & -1.0 & 0.5 \\ 0.5 & 1.5 & 1.0 \\ 3.0 & 2.0 & -1.5 \end{bmatrix} \qquad y_s = \begin{bmatrix} 1.0 \\ -1.0 \\ -1.0 \\ 1.0 \end{bmatrix}$$

- La Red Neuronal Multicapa se modela de la siguiente manera, la entrada X está compuesta por **[X1, X2, X3]**, los pesos se representan por **W1, W2 y W3**, los **bias** se representan por **b1, b2 y b3**, la salida se representa con $\hat{y}$ y la función de activación empleada es **tanh**.

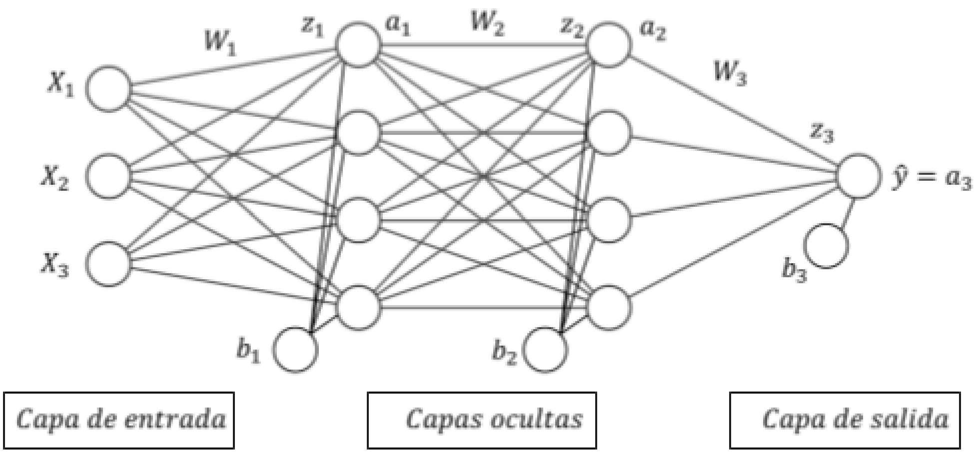
Además:

Primera capa
$z_1 = W_1 \cdot X + b_1$
$\mathbf{a}_1 = \tanh(z_1)$

Segunda capa
$z_2 = W_2 \cdot \mathbf{a}_1 + b_2$
$\mathbf{a}_2 = \tanh(z_2)$

Tercera capa
$z_3 = W_3 \cdot \mathbf{a}_2 + b_3$
$\mathbf{a}_3 = \tanh(z_3)$
$\mathbf{a}_3 = \hat{y}$

Arquitectura de la Red Neuronal



- **Las matrices de pesos y bias** se inicializan aleatoriamente, pero para que los resultados coincidan con los resultados haciendo uso de la librería Micrograd, se tomarán los siguientes valores:

$$W_1 = \begin{bmatrix} -0.341 & 0.488 & -0.579 \\ -0.741 & -0.542 & -0.716 \\ -0.432 & -0.389 & -0.726 \\ 0.854 & -0.244 & -0.297 \end{bmatrix} \qquad W_2 = \begin{bmatrix} 0.824 & -0.793 & 0.857 & 0.389 \\ -0.840 & 0.601 & -0.711 & -0.538 \\ -0.962 & -0.148 & 0.315 & 0.844 \\ 0.759 & -0.675 & -0.054 & 0.430 \end{bmatrix}$$

$$W_3 = \begin{bmatrix} -0.355 & -0.301 & 0.569 & -0.967 \end{bmatrix}$$

$$b_1 = \begin{bmatrix} 0.421 \\ -0.068 \\ 0.392 \\ 0.079 \end{bmatrix}$$

$$b_2 = \begin{bmatrix} 0.281 \\ 0.671 \\ 0.275 \\ 0.209 \end{bmatrix}$$

$$b_3 = \begin{bmatrix} 0.998 \end{bmatrix}$$

- Para ilustrar el proceso de cálculo manual de Forward propagation se utilizarán las entradas de la primera fila de la matriz de entradas Xs la cual se nombra como Xs1.

$$X_{s1} = \begin{bmatrix} 2.5 & 3.5 & -0.5 \end{bmatrix}$$

- Para el cálculo de Z1 se utilizará a la matriz W1, el bias b1 y la traspuesta de Xs1:

$$z_1 = \begin{bmatrix} -0.341 & 0.488 & -0.579 \\ -0.741 & -0.542 & -0.716 \\ -0.432 & -0.389 & -0.726 \\ 0.854 & -0.244 & -0.297 \end{bmatrix} . \begin{bmatrix} 2.5 \\ 3.5 \\ -0.5 \end{bmatrix} + \begin{bmatrix} 0.421 \\ -0.068 \\ 0.392 \\ 0.079 \end{bmatrix}$$

Se obtiene:

$$z_1 = \begin{bmatrix} 1.564 \\ -3.459 \\ -1.685 \\ 1.508 \end{bmatrix}$$

Entonces a1 es:

$$a_1 = \begin{bmatrix} 0.916 \\ -0.998 \\ -0.934 \\ 0.907 \end{bmatrix}$$

- Asimismo, calculamos z2, a2, z3 y a3. A continuación los resultados logrados:

z1	a1	z2	a2	z3	a3
[1.564, -3.459, -1.685, 1.508]	[0.916, -0.998, -0.934,0.907]	[1.380, -0.522, 0.013, 2.019]	[0.881, -0.479, 0.013, 0.965]	[-0.096]	[-0.096]

- Como se ha utilizado un enfoque de **Batch Gradient Descent**, se necesita realizar Forward para cada una de las entradas, lo que mostraremos en la siguiente tabla. Por simplicidad, utilizaremos la misma notación de variables que hemos utilizado anteriormente.

z1	$\begin{bmatrix} 1.564 & -1.721 & 0.403 & 1.241 \\ -3.459 & -2.847 & -1.967 & -2.300 \\ -1.685 & -1.309 & -1.133 & -0.591 \\ 1.508 & 3.592 & -0.158 & 2.599 \end{bmatrix}$
a1	$\begin{bmatrix} 0.916 & -0.938 & 0.382 & 0.846 \\ -0.998 & -0.993 & -0.962 & -0.980 \\ -0.934 & -0.864 & -0.812 & -0.531 \\ 0.907 & 0.998 & -0.157 & 0.989 \end{bmatrix}$
z2	$\begin{bmatrix} 1.380 & -0.056 & 0.601 & 1.685 \\ -0.522 & 0.939 & 0.433 & -0.783 \\ 0.013 & 1.895 & -0.338 & 0.274 \\ 2.019 & 0.644 & 1.125 & 1.967 \end{bmatrix}$
a2	$\begin{bmatrix} 0.881 & -0.056 & 0.538 & 0.933 \\ -0.479 & 0.735 & 0.408 & -0.654 \\ 0.013 & 0.956 & -0.326 & 0.268 \\ 0.965 & 0.568 & 0.809 & 0.962 \end{bmatrix}$
z3	[[-0.096, 0.792, -0.284, 0.087]]
a3	[[-0.096, 0.659, -0.276, 0.087]]

- Luego en el proceso **Backward propagation**, hemos utilizado el error cuadrático medio como función de pérdida:

$$\text{Loss} = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2$$

- Ahora actualizaremos los pesos y bias con el objetivo de **minimizar la pérdida**. Para ello utilizaremos el descenso de gradiente que requiere de establecer una tasa de aprendizaje n.

$$w' = w - \eta \frac{\partial Error}{\partial w} \qquad b' = b - \eta \sum_{i=1}^{n} \frac{\partial Error}{\partial b}$$

- El **error total** estaría dado por la suma de todas las pérdidas. Por lo tanto:

$$\text{Error} = \sum_{i=1}^{n} \left(\hat{y}_i - y_i \right)^2$$

Para calcular los gradientes respecto a los pesos y bias se utiliza la regla de la cadena, resultando:

$$\frac{\partial Error}{\partial w} = \frac{\partial Error}{\partial y} . \frac{\partial y}{\partial z} . \frac{\partial z}{\partial x} = 2.(\hat{y} - y) .(1 - tanh^2(z))) .x$$

$$\frac{\partial Error}{\partial b} = \frac{\partial Error}{\partial y} . \frac{\partial y}{\partial z} . \frac{\partial z}{\partial b} = 2.(\hat{y} - y) .(1 - tanh^2(z)))$$

- La propagación del error también implica determinar el delta de cada capa:

$$\delta_3 = 2.(\hat{y} - y) .(1 - tanh^2(z_3))$$

$$\delta_2 = W_3 . \delta_3 .(1 - tanh^2(z_2))$$

$$\delta_1 = W_2 . \delta_2 .(1 - tanh^2(z_1))$$

- Con una tasa de aprendizaje de 0.1, se procede a mostrar el cálculo de la actualización de los pesos y bias.

$$W_1' = W_1 - 0.1\delta_1.X_s$$

$$W_2' = W_2 - 0.1\delta_2.a_1$$

$$W_3' = W_3 - 0.1\delta_3.a_2$$

$$b_1' = b_1 - 0.1\sum \delta_1$$

$$b_2' = b_2 - 0.1\sum \delta_2$$

$$b_3' = b_3 - 0.1\sum \delta_3$$

- Los resultados de los cálculos los mostramos a continuación:

δ_1	$\begin{bmatrix} 0.16346314 & -0.16233431 & -0.84395226 & 0.23575742 \\ 0.00099784 & 0.01586053 & 0.01998641 & 0.00689013 \\ -0.07805895 & -0.07299838 & 0.06407392 & -0.33038147 \\ -0.21089959 & -0.00172385 & 0.42081974 & -0.01938593 \end{bmatrix}$
δ_2	$\begin{bmatrix} 0.17247053 & -0.66320567 & -0.33710103 & 0.08273963 \\ 0.50348499 & -0.25975528 & -0.33555434 & 0.31225281 \\ -1.2354633 & 0.09230852 & 0.68023774 & -0.95796281 \\ 0.14300552 & -1.2285945 & -0.44579195 & 0.13199941 \end{bmatrix}$
δ_3	$\begin{bmatrix} -2.17103598 & 1.87566244 & 1.33716536 & -1.81311873 \end{bmatrix}$

W'_1	$\begin{bmatrix} -0.34554259 & 0.4935403 & -0.44262008 \\ -0.7502685 & -0.54508613 & -0.71749508 \\ -0.28692567 & -0.3123435 & -0.78227928 \\ 0.8926412 & -0.22996956 & -0.35290302 \end{bmatrix}$
W'_2	$\begin{bmatrix} 0.75195582 & -0.86552398 & 0.79253729 & 0.42652176 \\ -0.92420782 & 0.62329247 & -0.69687519 & -0.59349348 \\ -0.78492013 & -0.29102007 & 0.2121559 & 1.05201748 \\ 0.63649636 & -0.81285838 & -0.17622029 & 0.51966873 \end{bmatrix}$
W'_3	$\begin{bmatrix} -0.05558177 & -0.71627873 & 0.48484404 & -0.79736761 \end{bmatrix}$
b'_1	$\begin{bmatrix} 0.48137072 \\ -0.07279767 \\ 0.43346154 \\ 0.05973635 \end{bmatrix}$
b'_2	$\begin{bmatrix} 0.3552724 \\ 0.64854159 \\ 0.41713561 \\ 0.34912214 \end{bmatrix}$
b'_3	$\begin{bmatrix} 1.07507519 \end{bmatrix}$

Pregunta 2.- Verificar el resultado del cálculo a mano usando la librería vista en la segunda clase (Micrograd). Hacer lo mismo usando PyTorch.

- **Micrograd** es una librería que **define una clase Value** que se usa para representar valores numéricos en una red de cómputo que admite operaciones como suma, multiplicación y otras funciones, junto con la capacidad de calcular gradientes para el aprendizaje automático mediante la diferenciación automática.

```
#%%
import math
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
#%%
class Value:

 def __init__(self, data, _children=(), _op='', label=''):
  self.data = data
  self.grad = 0.0
  self._backward = lambda: None
  self._prev = set(_children)
  self._op = _op
  self.label = label

 def __repr__(self):
  return f"Value(data={self.data}, grad={self.grad})"

 def __add__(self, other):
  other = other if isinstance(other, Value) else Value(other)
  out = Value(self.data + other.data, (self, other), '+')

  def _backward():
   self.grad += 1.0 * out.grad
   other.grad += 1.0 * out.grad
  out._backward = _backward

  return out

 def __mul__(self, other):
  other = other if isinstance(other, Value) else Value(other)
  out = Value(self.data * other.data, (self, other), '*')

  def _backward():
   self.grad += other.data * out.grad
   other.grad += self.data * out.grad
  out._backward = _backward

  return out
```

```
def __pow__(self, other):
  assert isinstance(other, (int, float)), "only supporting int/float powers for now"
  out = Value(self.data**other, (self,), f'**{other}')

  def _backward():
      self.grad += other * (self.data ** (other - 1)) * out.grad
  out._backward = _backward

  return out

def __rmul__(self, other): # other * self
  return self * other

def __truediv__(self, other): # self / other
  return self * other**-1

def __neg__(self): # -self
  return self * -1

def __sub__(self, other): # self - other
  return self + (-other)

def __radd__(self, other): # other + self
  return self + other

def tanh(self):
  x = self.data
  t = (math.exp(2*x) - 1)/(math.exp(2*x) + 1)
  out = Value(t, (self, ), 'tanh')

  def _backward():
    self.grad += (1 - t**2) * out.grad
  out._backward = _backward

  return out

def exp(self):
  x = self.data
  out = Value(math.exp(x), (self, ), 'exp')

  def _backward():
    self.grad += out.data * out.grad # NOTE: in the video I incorrectly used = instead of +=. Fixed here.
  out._backward = _backward

  return out


def backward(self):

  topo = []
  visited = set()
  def build_topo(v):
```

```
    if v not in visited:
      visited.add(v)
      for child in v._prev:
        build_topo(child)
      topo.append(v)
  build_topo(self)

  self.grad = 1.0
  for node in reversed(topo):
    node._backward()
#%%
import random
class Neuron:

 def __init__(self, nin):
  self.w = [Value(random.uniform(-1,1)) for _ in range(nin)]
  print("W:", self.w)
  self.b = Value(random.uniform(-1,1))
  print("b:", self.b)

 def __call__(self, x):
  # w * x + b
  act = sum((wi*xi for wi, xi in zip(self.w, x)), self.b)
  out = act.tanh()
  return out

 def parameters(self):
  return self.w + [self.b]

class Layer:

 def __init__(self, nin, nout):
  self.neurons = [Neuron(nin) for _ in range(nout)]

 def __call__(self, x):
  outs = [n(x) for n in self.neurons]
  return outs[0] if len(outs) == 1 else outs

 def parameters(self):
  return [p for neuron in self.neurons for p in neuron.parameters()]

class MLP:

 def __init__(self, nin, nouts):
  sz = [nin] + nouts
  self.layers = [Layer(sz[i], sz[i+1]) for i in range(len(nouts))]

 def __call__(self, x):
  for layer in self.layers:
    x = layer(x)
  return x
```

```
 def parameters(self):
   return [p for layer in self.layers for p in layer.parameters()]
#%%
n = MLP(3, [4, 4, 1])
#%%
xs = [
  [2.5, 3.5, -0.5],  # Primera muestra
  [4.0, -1.0, 0.5],  # Segunda muestra
  [0.5, 1.5, 1.0],   # Tercera muestra
  [3.0, 2.0, -1.5]   # Cuarta muestra
]
ys = [1.0, -1.0, -1.0, 1.0]
#%%
for k in range(5):  # Incrementa las iteraciones para ver más cambios

  # forward pass
  ypred = [n(x) for x in xs]
  loss = sum((yout - ygt)**2 for ygt, yout in zip(ys, ypred))

  # Imprimir los valores de predicciones y pérdida (loss)
  print(f"Iteración {k}")
  print("Predicciones:", [yp.data for yp in ypred])
  print("Loss:", loss.data)

  # backward pass
  for p in n.parameters():
      p.grad = 0.0
  loss.backward()

  # Imprimir gradientes antes de la actualización
  print("Gradientes:")
  for i, p in enumerate(n.parameters()):
      print(f"Parámetro {i}: {p.data}, Gradiente: {p.grad}")

  # update
  for p in n.parameters():
      p.data += -0.1 * p.grad

  # Imprimir valores actualizados
  print("Pesos y bias actualizados:")
  for i, p in enumerate(n.parameters()):
      print(f"Parámetro {i}: {p.data}")
  print("-" * 50)
#%%
```

- En la primera iteración se obtuvieron los siguientes resultados que coinciden con los mostrados en el cálculo manual:

Iteración 0

Predicciones:

[-0.09551180109351012, 0.6594294380423391, -0.2762009023561765, 0.08659177780839312]

Pesos y bias actualizados:
Parámetro 0: -0.3455425864395133
Parámetro 1: 0.49354030472046384
Parámetro 2: -0.44262007762858524
Parámetro 3: 0.4813707157639071
Parámetro 4: -0.7502685003738252
Parámetro 5: -0.5450861251164729
Parámetro 6: -0.7174950822443538
Parámetro 7: -0.07279766624127787
Parámetro 8: -0.2869256709509327
Parámetro 9: -0.3123434983594058
Parámetro 10: -0.7822792789756129
Parámetro 11: 0.43346153792328573
Parámetro 12: 0.8926412032411102
Parámetro 13: -0.22996956288809678
Parámetro 14: -0.35290302011138436
Parámetro 15: 0.0597363504502513
Parámetro 16: 0.7519558171679739
Parámetro 17: -0.865523978704413
Parámetro 18: 0.7925372877736636
Parámetro 19: 0.4265217627334761
Parámetro 20: 0.3552723998299817
Parámetro 21: -0.9242078198017564
Parámetro 22: 0.623292469237169
Parámetro 23: -0.696875192571201
Parámetro 24: -0.593493477673018
Parámetro 25: 0.6485415878541207
Parámetro 26: -0.7849201299364432
Parámetro 27: -0.2910200672775243
Parámetro 28: 0.21215590477957946
Parámetro 29: 1.0520174753370677
Parámetro 30: 0.4171356087187202
Parámetro 31: 0.6364963596437079
Parámetro 32: -0.8128583812158143
Parámetro 33: -0.17622028514159255
Parámetro 34: 0.5196687337910336
Parámetro 35: 0.3491221402067136
Parámetro 36: -0.055581770552016574
Parámetro 37: -0.7162787275772213
Parámetro 38: 0.48484404322738217
Parámetro 39: -0.797367614376691
Parámetro 40: 1.075075192880589

- Luego implementamos un MLP en Pytorch:

```
#%%
import torch
import torch.nn as nn
import numpy as np
#%%
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
#%%
# Pesos (W) y sesgos (b) para la primera capa
W1_np = np.array([
    [-0.34108091502837, 0.4875444774924518, -0.5786687891714137],
    [-0.7406084700295616, -0.5419469493214886, -0.7157868250555495],
    [-0.4315505068114813, -0.3888294993216346, -0.7260616380268912],
    [0.8544519713956533, -0.24436625968891335, -0.29745436944045234]
])
b1_np = np.array([
    [0.42066411352123567],
    [-0.06842417572242421],
    [0.3917250492343052],
    [0.07861738679252617]
])

# Pesos (W) y sesgos (b) para la segunda capa
W2_np = np.array([
    [0.8240668394757076, -0.7925535106179675, 0.856718957818575, 0.38940887530348656],
    [-0.8401491947426882, 0.6005090770290116, -0.710761829952594, -0.5376371931922113],
    [-0.9617475006414558, -0.1484109682541901, 0.315131084013728, 0.843811249537862],
    [0.7589476929906058, -0.6751635852663345, -0.05422983414165872, 0.43000805343850246]
])
b2_np = np.array([
    [0.2807627460701372],
    [0.6705844066805486],
    [0.2750476238785533],
    [0.20918398742213418]
])

# Pesos (W) y sesgos (b) para la capa de salida
W3_np = np.array([
    [-0.3547048938594579, -0.3011280508308256, 0.5691627652457749, -0.9665772078201607]
])
b3_np = np.array([[0.9979425023631463]])

# Convertimos a tensores de torch
W1 = torch.tensor(W1_np, dtype=torch.float32)
b1 = torch.tensor(b1_np, dtype=torch.float32)
W2 = torch.tensor(W2_np, dtype=torch.float32)
b2 = torch.tensor(b2_np, dtype=torch.float32)
W3 = torch.tensor(W3_np, dtype=torch.float32)
b3 = torch.tensor(b3_np, dtype=torch.float32)
#%%
X = torch.tensor([[2.5, 3.5, -0.5],
                  [4.0, -1.0, 0.5],
```

```
              [0.5, 1.5, 1.0],
              [3.0, 2.0, -1.5]], dtype=torch.float32).to(device)
y = torch.tensor([[1.0], [-1.0], [-1.0], [1.0]], dtype=torch.float32).to(device)
#%%
class MLP(nn.Module):
  def __init__(self, W1, b1, W2, b2, W3, b3):
      super(MLP, self).__init__()
      # Definimos las capas lineales
      self.layer1 = nn.Linear(3, 4)
      self.layer2 = nn.Linear(4, 4)
      self.output_layer = nn.Linear(4, 1)

      # Asignamos los pesos y bias iniciales desde numpy
      with torch.no_grad():
          self.layer1.weight = nn.Parameter(W1)
          self.layer1.bias = nn.Parameter(b1)
          self.layer2.weight = nn.Parameter(W2)
          self.layer2.bias = nn.Parameter(b2)
          self.output_layer.weight = nn.Parameter(W3)
          self.output_layer.bias = nn.Parameter(b3)

  def forward(self, x):
      x = torch.tanh(self.layer1(x))
      x = torch.tanh(self.layer2(x))
      x = torch.tanh(self.output_layer(x))
      return x
#%%
model = MLP(W1, b1, W2, b2, W3, b3).to(device)
#%%
# Definir la función de pérdida (MSE) y el optimizador (SGD)
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
#%%
# Paso de entrenamiento (forward + backward)
def train_step(X, y):
  optimizer.zero_grad()         # Limpiar los gradientes previos
  y_pred = model(X)             # Forward propagation
  loss = criterion(y_pred, y)   # Calcular la pérdida
  loss.backward()               # Retropropagación
  optimizer.step()              # Actualización de los pesos
  return loss.item()
#%%
# Mostrar los pesos y bias iniciales
print("Pesos y bias iniciales:")
for name, param in model.named_parameters():
  print(f"{name}: {param.data}")
#%%
# Realizar un paso de entrenamiento
loss = train_step(X, y)

# Mostrar los pesos y bias después del entrenamiento
print("\nPesos y bias después de la retropropagación:")
```

```
for name, param in model.named_parameters():
  print(f"{name}: {param.data}")

# Mostrar la pérdida final
print(f"\nPérdida final: {loss}")
#%%
```

- Los resultados obtenidos discrepan ligeramente de los calculados con los métodos anteriores:

Pesos y bias después de la retropropagación:

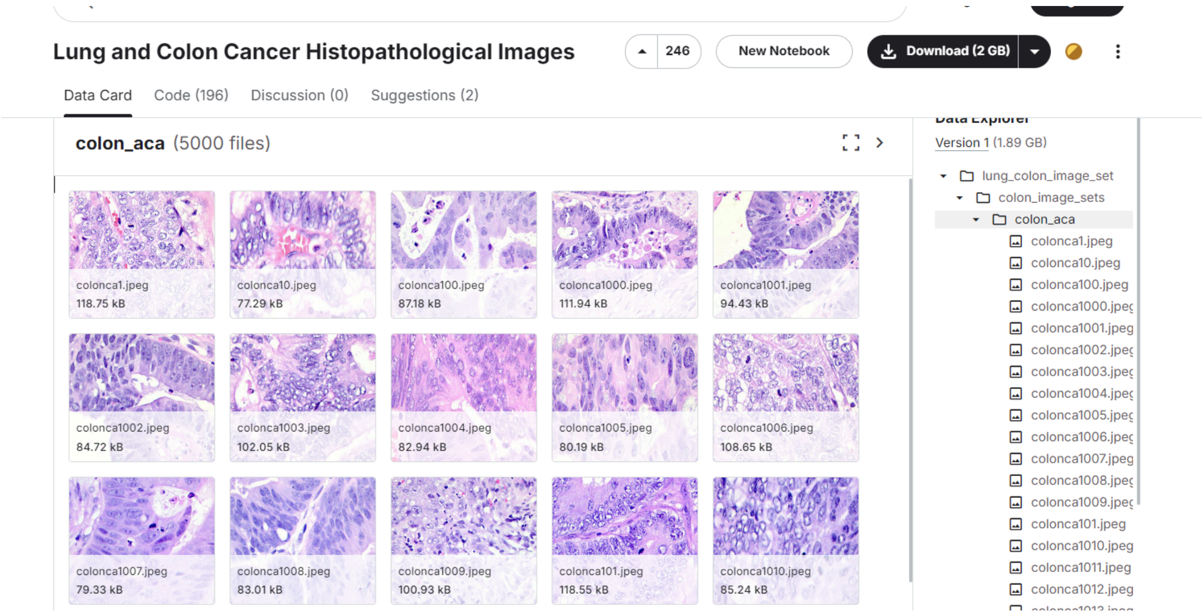
```
layer1.weight: tensor([[-0.3652,  0.4924, -0.5307],
        [-0.7427, -0.5432, -0.7167],
        [-0.4137, -0.3822, -0.7352],
        [ 0.8516, -0.2657, -0.3195]], device='cuda:0')

layer1.bias: tensor([[ 0.4201],
        [-0.0663],
        [ 0.3961],
        [ 0.0718]], device='cuda:0')

layer2.weight: tensor([[ 0.8030, -0.8150,  0.8352,  0.4034],
        [-0.8521,  0.5926, -0.7187, -0.5403],
        [-0.9284, -0.1723,  0.2961,  0.8872],
        [ 0.7380, -0.7009, -0.0788,  0.4464]], device='cuda:0')

layer2.bias: tensor([[0.2956],
        [0.7166],
        [0.2862],
        [0.2188]], device='cuda:0')

output_layer.weight: tensor([[-0.3148, -0.4109,  0.5292, -0.9650]], device='cuda:0')
output_layer.bias: tensor([[0.9839]], device='cuda:0')
```

Pérdida final: 1.1816823482513428

PREGUNTA 3: Usando la data de Kaggle sobre Lung Cancer (Histopathological Images),clasificar dado la imagen está en uno de estas clases: ['adenocarcinoma', 'benign', 'squamous cell carcinoma'].

Descripción del DataSet

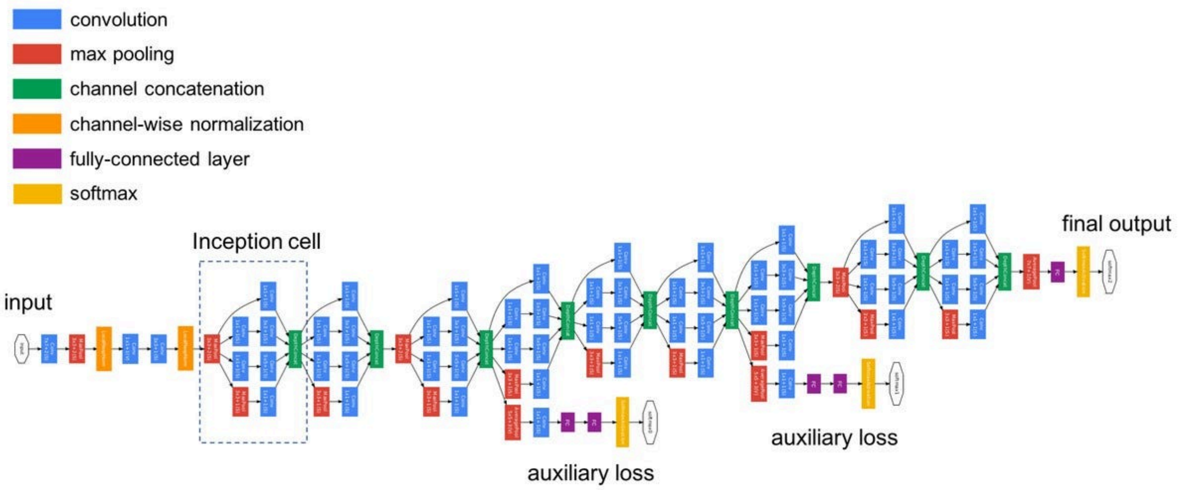
El conjunto de datos contiene 15000 imágenes en color, divididas en 3 clases de 5000 imágenes cada una. Todas las imágenes tienen un tamaño de 768 x 768 píxeles y están en formato jpeg. Las clases son adenocarcinomas de pulmón, carcinomas de células escamosas de pulmón y tejidos pulmonares benignos.



Para el entrenamiento, se separaron imágenes en carpetas de train, validation y test con cantidades del **75%, 15% y 15%** del dataset respectivamente.
Se eligieron 3 arquitecturas para el entrenamiento de los modelos: Inception v1, Resnet50 y Inception-resnet v2.

1. Arquitectura Inception V1

Se procedió a implementar la arquitectura Inception V1 para el problema de clasificación de imágenes Lung Cancer en local: *Pregunta3_Inceptionv1.ipynb*



- **Convolución (azul):** Aplica filtros para detectar características como bordes y texturas.
- **Max Pooling (rojo):** Reduce el tamaño de la imagen, manteniendo la información más importante.
- **Channel Concatenation (verde):** Combina salidas de varias convoluciones y pooling para obtener una representación más diversa.
- **Normalización (amarillo):** Ajusta los valores de los canales para hacerlos comparables y estabilizar el entrenamiento.
- **Fully-connected Layer (morado):** Usa todas las características extraídas para hacer una predicción.
- **Softmax (amarillo):** Convierte la salida en probabilidades para la clasificación.
- **Inception Cell:** Combina filtros de diferentes tamaños (1x1, 3x3, 5x5) y pooling, capturando detalles en diferentes escalas.
- **Auxiliary Loss:** Dos salidas auxiliares que ayudan al entrenamiento al proporcionar retroalimentación intermedia.
- **Salida final:** Resultado de la clasificación.

1.1. Parámetros usados para el entrenamiento

A continuación, se detalla la descripción de cada capa y los parámetros involucrados:

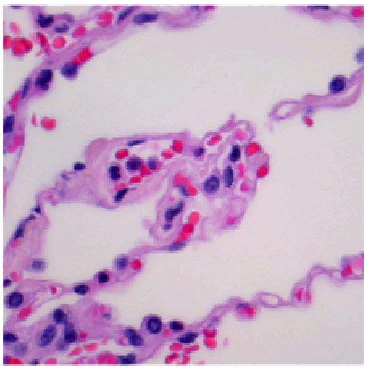
Capa	Fórmula de los Parámetros	Total de Parámetros
Convolution 1	7 × 7 × 3 × 64	9,408
Inception Block 1	(1 × 1 × 64 + 3 × 3 × 64 + 5 × 5 × 64) × 128	887,040
Inception Block 2	(1 × 1 × 128 + 3 × 3 × 128 + 5 × 5 × 128) × 256	3,548,160
Inception Block 3	(1 × 1 × 256 + 3 × 3 × 256 + 5 × 5 × 256) × 512	14,192,640
Inception Block 4	(1 × 1 × 512 + 3 × 3 × 512 + 5 × 5 × 512) × 1024	56,619,008
Inception Block 5	(1 × 1 × 1024 + 3 × 3 × 1024 + 5 × 5 × 1024) × 2048	226,416,640
Fully Connected	2048 × 3	6,144
Final Output	Softmax	0

1.2. Resultados

- La red neuronal se entrenó con **10 épocas**, obteniendo una **precisión de 97.24%**, luego se procedió a probar una imágen dando como resultado la clase de **benign** a un 100% de confidencialidad.

```
Epoch [1/10], Loss: 0.4093
Validation Loss: 0.1961, Accuracy: 92.98%
Epoch [2/10], Loss: 0.2671
Validation Loss: 0.2879, Accuracy: 87.20%
Epoch [3/10], Loss: 0.2284
Validation Loss: 0.1590, Accuracy: 92.93%
Epoch [4/10], Loss: 0.1980
Validation Loss: 0.3810, Accuracy: 81.11%
Epoch [5/10], Loss: 0.1699
Validation Loss: 0.1200, Accuracy: 95.20%
Epoch [6/10], Loss: 0.1561
Validation Loss: 0.1104, Accuracy: 95.33%
Epoch [7/10], Loss: 0.1504
Validation Loss: 0.1222, Accuracy: 94.62%
Epoch [8/10], Loss: 0.1348
Validation Loss: 0.1032, Accuracy: 95.78%
Epoch [9/10], Loss: 0.1220
Validation Loss: 0.0947, Accuracy: 96.00%
Epoch [10/10], Loss: 0.1131
Validation Loss: 0.0743, Accuracy: 97.24%
Evaluando el modelo en el conjunto de test...
Validation Loss: 0.0687, Accuracy: 97.69%
```

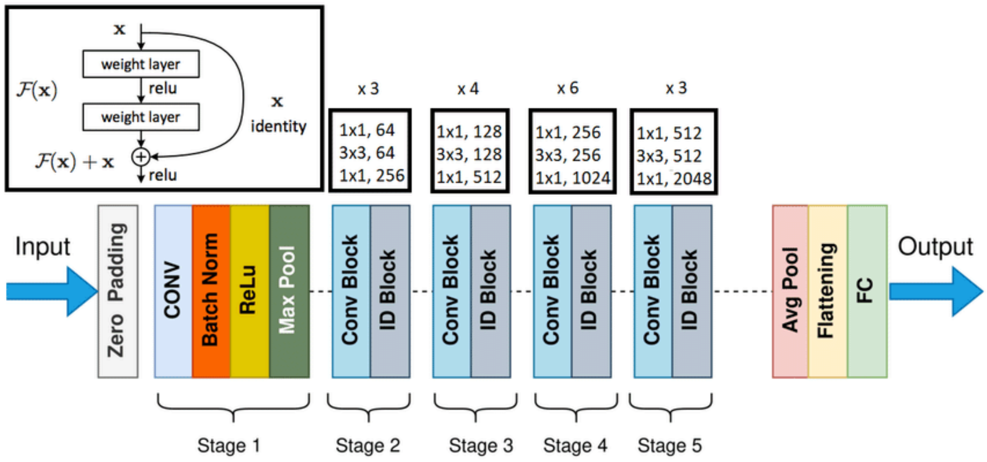
```
image_path = './LungCancer/test/benign/0087.jpg'
predicted_class, confidence, probabilities = predict_image(image_path, model)

print(f'Predicción: {classes[predicted_class]} con {confidence*100:.2f}% confidence')
```



```
Predicción: benign con 100.00% confidence
```

2. Arquitectura ResNet

Se procedió a implementar la arquitectura ResNet para el problema de clasificación de imágenes Lung Cancer en GoogleColab: *Pregunta3_ResNet.ipynb*



- **Zero Padding:** Se añaden ceros en los bordes de la imagen para mantener las dimensiones originales antes de aplicar convoluciones.

- **Capa Convolucional (CONV)**: Se aplica una convolución con filtro 7×7 y stride de 2, extrayendo características iniciales y reduciendo el tamaño de la imagen.

- **Batch Normalization:** Normaliza las activaciones para acelerar y estabilizar el entrenamiento.

- **ReLU:** Función de activación que introduce no linealidad, anulando valores negativos y dejando pasar solo los positivos.

- **Max Pooling:** Reduce las dimensiones espaciales seleccionando el valor máximo en ventanas de 3×3 y stride de 2.

- **Bloques Residuales:**

    - **Conv Block:** Realiza convoluciones con filtros 1×1, 3×3 , ajustando las dimensiones de las características.
    - **Identity Block:** Similar al Conv Block, pero sin modificar las dimensiones, permitiendo conexiones de salto directo.

- **Average Pooling:** Reduce cada canal a un solo valor promedio, comprimiendo la información.

- **Flattening:** Convierte la salida bidimensional en un vector unidimensional para las siguientes capas.

- **Fully Connected (FC):** Realiza la clasificación final utilizando la información extraída, generalmente seguida de una función softmax para obtener las probabilidades de cada clase.

2.1. Parámetros usados para el entrenamiento

A continuación, se detalla la descripción de cada capa y los parámetros involucrados:

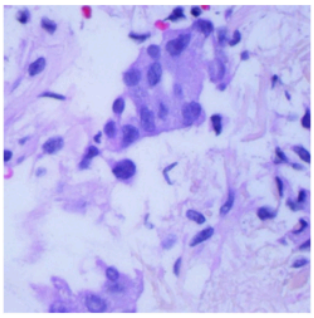
Capa	Fórmula de los Parámetros	Total de Parámetros
Convolution 1	7 × 7 × 3 × 64	9,408
Max Pooling	Pooling	0
Residual Block 1	(1 × 1 × 64 + 3 × 3 × 64 + 1 × 1 × 256) × 3	214,272
Residual Block 2	(1 × 1 × 128 + 3 × 3 × 128 + 1 × 1 × 512) × 4	1,180,160
Residual Block 3	(1 × 1 × 256 + 3 × 3 × 256 + 1 × 1 × 1024) × 6	6,706,176
Residual Block 4	(1 × 1 × 512 + 3 × 3 × 512 + 1 × 1 × 2048) × 3	13,421,056
Fully Connected	2048 × 3	6,144
Final Output	Softmax	0

2.2 Resultados

La red se entrenó con **10 épocas**, obteniendo una **precisión de 98%**, luego se procedió a probar una imágen dando como resultado la clase de **adenocarcinoma** a un 98.48% de confidencialidad.

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.
  warnings.warn(_create_warning_msg(
Epoch [1/10], Loss: 0.3610, Accuracy: 86.28%
Validation Accuracy: 81.07%
Epoch [2/10], Loss: 0.2721, Accuracy: 89.44%
Validation Accuracy: 89.51%
Epoch [3/10], Loss: 0.2294, Accuracy: 90.74%
Validation Accuracy: 90.89%
Epoch [4/10], Loss: 0.1994, Accuracy: 91.89%
Validation Accuracy: 90.93%
Epoch [5/10], Loss: 0.2032, Accuracy: 91.87%
Validation Accuracy: 95.69%
Epoch [6/10], Loss: 0.1891, Accuracy: 92.97%
Validation Accuracy: 88.40%
Epoch [7/10], Loss: 0.2003, Accuracy: 92.85%
Validation Accuracy: 95.42%
Epoch [8/10], Loss: 0.1381, Accuracy: 94.67%
Validation Accuracy: 96.89%
Epoch [9/10], Loss: 0.1226, Accuracy: 95.29%
Validation Accuracy: 97.42%
Epoch [10/10], Loss: 0.1182, Accuracy: 95.98%
Validation Accuracy: 98.00%
Accuracy on test set: 98.00%
```

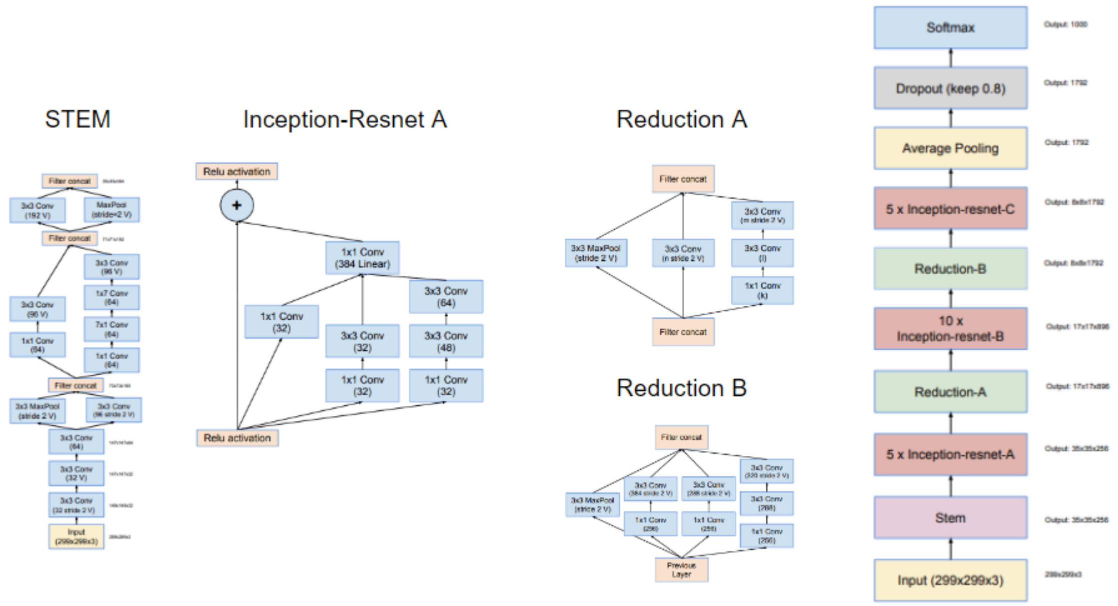
```
image_path = './data/test/adenocarcinoma/0096.jpg'
predicted_class, confidence, probabilities = predict_image(image_path, model)

classes = ['adenocarcinoma', 'benign', 'squamous_cell_carcinoma']

print(f'Predicción: {classes[predicted_class]} con {confidence*100:.2f}% confidence')
```



```
Predicción: adenocarcinoma con 98.48% confidence
```

3. Arquitectura Inception + ResNetV2

Se procedió a implementar la arquitectura Inception-ResNetV2 para el problema de clasificación de imágenes Lung Cancer en GoogleColab: *Pregunta3_Inception-Resnetv2.ipynb*



- **STEM:** Bloque inicial con convoluciones 3x3 y *Max Pooling* para reducir la resolución de la imagen, manteniendo características importantes.
- **Inception-ResNet-A/B/C:** Bloques que aplican convoluciones en paralelo (1x1, 3x3) y luego suman la salida con la entrada original (conexión residual). Extraen características a diferentes escalas.
- **Reduction-A/B:** Bloques que reducen la resolución de las características con *Max Pooling* y convoluciones 3x3, luego concatenan las salidas.
- **Average Pooling y Softmax:** Tras pasar por todos los bloques, se realiza un *Average Pooling* y se usa *Softmax* para la clasificación final.
- **Conexiones residuales:** Mejoran el flujo de gradientes y el entrenamiento de redes profundas, evitando la pérdida de información.

3.1. Parámetros usados para el entrenamiento

A continuación, se detalla la descripción de cada capa y los parámetros involucrados:

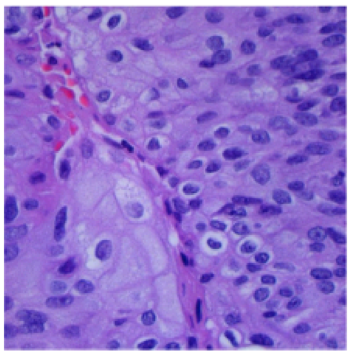
Capa	Fórmula de los Parámetros	Total de Parámetros
Convolution 1	$7 \times 7 \times 3 \times 64$	9,408
Stem	$(1 \times 1 \times 64 + 3 \times 3 \times 64 + 3 \times 3 \times 192) \times 192$	110,784
Inception-ResNet Block A	$(1 \times 1 \times 32 + 3 \times 3 \times 32 + 1 \times 1 \times 256) \times 5$	1,536,000
Inception-ResNet Block B	$(1 \times 1 \times 128 + 1 \times 7 \times 128 + 1 \times 1 \times 896) \times 10$	6,451,200
Inception-ResNet Block C	$(1 \times 1 \times 192 + 1 \times 3 \times 192 + 1 \times 1 \times 1792) \times 5$	8,601,600
Reduction Block A	$(3 \times 3 \times 192 + 3 \times 3 \times 384 + 1 \times 1 \times 1024)$	3,411,968
Reduction Block B	$(1 \times 1 \times 256 + 1 \times 3 \times 256 + 3 \times 3 \times 1024)$	5,243,392
Fully Connected	1536×3	4,608
Final Output	Softmax	0

3.2. Resultados

La red neuronal se entrenó con **10 épocas,** obteniendo una **precisión de 96.62%**, luego se procedió a probar una imágen dando como resultado la clase **desquamous_cell_carcinoma** a un 95.87% de confidencialidad.

```
Epoch [1/10], Loss: 0.3566, Accuracy: 87.61%
Validation Accuracy: 86.36%
Epoch [2/10], Loss: 0.2733, Accuracy: 90.17%
Validation Accuracy: 93.69%
Epoch [3/10], Loss: 0.2183, Accuracy: 91.38%
Validation Accuracy: 94.58%
Epoch [4/10], Loss: 0.1996, Accuracy: 92.16%
Validation Accuracy: 88.09%
Epoch [5/10], Loss: 0.2276, Accuracy: 91.74%
Validation Accuracy: 87.87%
Epoch [6/10], Loss: 0.2607, Accuracy: 89.30%
Validation Accuracy: 91.02%
Epoch [7/10], Loss: 0.1959, Accuracy: 92.40%
Validation Accuracy: 96.80%
Epoch [8/10], Loss: 0.1505, Accuracy: 94.41%
Validation Accuracy: 96.53%
Epoch [9/10], Loss: 0.1468, Accuracy: 94.60%
Validation Accuracy: 97.60%
Epoch [10/10], Loss: 0.1387, Accuracy: 94.82%
Validation Accuracy: 97.16%
Accuracy on test set: 96.62%
```

```
image_path = './data/test/squamous_cell_carcinoma/0122.jpg'
predicted_class, confidence, probabilities = predict_image(image_path, model)

print(f'Predicción: {classes[predicted_class]} con {confidence*100:.2f}% confidence')
```



Predicción: squamous_cell_carcinoma con 95.87% confidence

4. Conclusiones

- **ResNet 50** mostró el mejor rendimiento general con una precisión de entrenamiento del 98% y una precisión de prueba también del **98%**, lo que indica una excelente capacidad para generalizar en nuevos datos.

- **Inception v1** tuvo un rendimiento ligeramente menor, con una precisión de entrenamiento de 97.69% y una precisión de prueba de **97.24%**, lo que demuestra que sigue siendo un modelo competitivo, aunque marginalmente superado por ResNet 50.

- **Inception-ResNet v2** alcanzó una precisión de entrenamiento más baja (94.82%), pero aún logró una precisión de prueba del **96.62%**, lo que indica que, aunque fue el modelo que más le costó ajustarse a los datos de entrenamiento, logró una buena generalización.

ARQUITECTURA	N° de Épocas	Precisión Train	Precisión Test	Entorno de entrenamiento
Inceptionv1	10	97.69 %	97.24%	Local
ResNet 50	10	98%	98%	Google Colab
Inception + Resnetv2	10	94.82%	96.62%	Google Colab

GPU Colab T4 16GB
GPU local: GTX 1060 6GB

Tiempo Google Colab: 2h
Tiempo Local: 5h