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

SECC. A

CURSO REDES NEURONALES Y APRENDIZAJE PROFUNDO

TAREA 1

INTEGRANTES GRUPO 8

Kevin Gómez Villanueva

Fernando Boza Gutarra

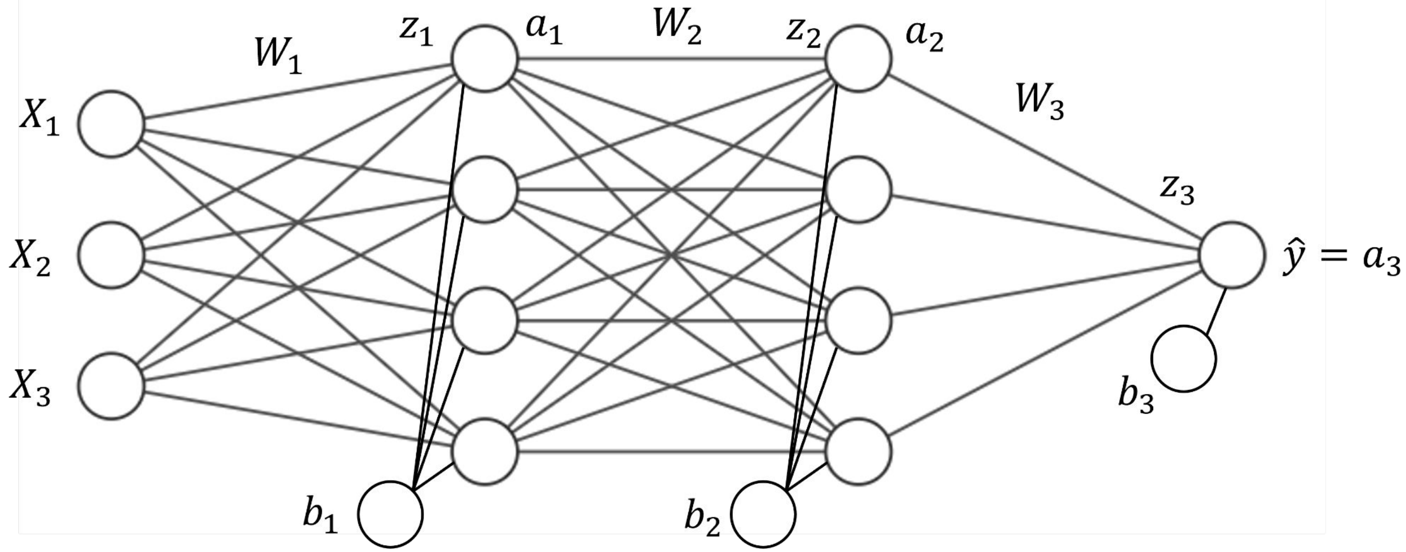
Umbert Lewis de la Cruz Rodriguez

Yovany Romero Ramos

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:**

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

Modelamiento de Red Neuronal



Capa de entrada *Capas ocultas* *Capa de salida*

Proceso ForWard

$$z_1 = W_1 . X + b_1 \qquad a_1 = tanh(z_1)$$

$$z_2 = W_2 . a_1 + b_2 \qquad a_2 = tanh(z_2)$$

$$z_3 = W_3 . a_2 + b_3 \qquad a_3 = tanh(z_3) = \hat{y}$$

Pesos y Bias

$$W_1 = \begin{bmatrix} -0.341 & 0.488 & -0.579 \\ -0.741 & -0.542 & -0.716 \\ -0.432 & -0.389 & -0.726 \\ 0.854 & -0.244 & -0.297 \end{bmatrix} \qquad b_1 = \begin{bmatrix} 0.421 \\ -0.068 \\ 0.392 \\ 0.079 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.824 & -0.793 & 0.857 & 0.389 \\ -0.840 & 0.601 & -0.711 & -0.538 \\ -0.962 & -0.148 & 0.315 & 0.844 \\ 0.759 & -0.675 & -0.054 & 0.430 \end{bmatrix} \qquad b_2 = \begin{bmatrix} 0.281 \\ 0.671 \\ 0.275 \\ 0.209 \end{bmatrix}$$

$$W_3 = \begin{bmatrix} -0.355 & -0.301 & 0.569 & -0.967 \end{bmatrix} \qquad b_3 = \begin{bmatrix} 0.998 \end{bmatrix}$$

Resultados Forward

z1	[[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]]
a1	[[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]]
z2	[[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]]
a2	[[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]]
z3	[[-0.096, 0.792, -0.284, 0.087]]
a3 = ypred	[[-0.096, 0.659, -0.276, 0.087]]

Proceso BackWard

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

$$w' = w - \eta\frac{\partial Error}{\partial w}$$

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

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

Proceso BackWard

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

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

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

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

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

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

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

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

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

Resultados BackWard

W'1	[[-0.34554259 0.4935403 -0.44262008] [-0.7502685 -0.54508613 -0.71749508] [-0.28692567 -0.3123435 -0.78227928] [0.8926412 -0.22996956 -0.35290302]]
W'2	[[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]]
W'3	[[-0.05558177 -0.71627873 0.48484404 -0.79736761]]
b'1	[[0.48137072] [-0.07279767] [0.43346154] [0.05973635]]
b'2	[[0.3552724] [0.64854159] [0.41713561] [0.34912214]]
b'3	[[1.07507519]]

Código

```python
# Capa Oculta 1
z1 = W1.dot(X.T) + b1
a1 = tanh(z1)
print("z1:", z1)
print("a1:", a1)
```

```
z1: [[ 1.56370189 -1.72053842  0.40277158  1.24051351]
 [-3.45886626 -2.84680452 -1.96743566 -2.30046325]
 [-1.68502365 -1.3086783  -1.13335609 -0.59149301]
 [ 1.50819259  3.59206435 -0.15816039  2.59942234]]
a1: [[ 0.91601815 -0.93792783  0.38231794  0.84560199]
 [-0.99802181 -0.99328777 -0.96165318 -0.98011465]
 [-0.93351042 -0.8639406  -0.81216475 -0.53096855]
 [ 0.90661778  0.99848409 -0.15685467  0.98901479]]
```

```python
# Capa Oculta 2
z2 = W2.dot(a1) + b2
a2 = tanh(z2)
print("z2:", z2)
print("a2:", a2)
```

```
z2: [[ 1.37989756 -0.0562545   0.60110234  1.68462892]
 [-0.52225653  0.93933922  0.43348545 -0.78275412]
 [ 0.01302297  1.89478974 -0.33821991  0.2744689 ]
 [ 2.0186989   0.64418511  1.1252113   1.96676804]]
a2: [[ 0.88092833 -0.05619523  0.5378335   0.93345966]
 [-0.47943973  0.73491851  0.40823003 -0.65428466]
 [ 0.01302223  0.95578918 -0.32588728  0.26777823]
 [ 0.96532513  0.56774241  0.80937396  0.96160293]]
```

```python
# Capa de Salida
z3 = W3.dot(a2) + b3
a3 = tanh(z3)
print("z3:", z3)
print("a3:", a3)
```

```
z3: [[-0.09580384  0.79180339 -0.28356452  0.08680918]]
a3: [[-0.0955118   0.65942944 -0.2762009   0.08659178]]
```

```python
error = a3 - y.T
delta3 = 2 * error * tanh_derivative(z3)
print("error:", error)
print("delta3:", delta3)
```

```
error: [[-1.0955118   1.65942944  0.7237991  -0.91340822]]
delta3: [[-2.17103598  1.87566244  1.33716536 -1.81311873]]
```

```python
loss = np.sum((y.T - a3)**2)
print(loss)
```

```
5.312051880293905
```

```python
delta2 = W3.T.dot(delta3) * tanh_derivative(z2)
print("delta2:", delta2)
```

```
delta2: [[ 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]]
```

```python
delta1 = W2.T.dot(delta2) * tanh_derivative(z1)
print("delta1:", delta1)
```

```
delta1: [[ 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]]
```

```python
learning_rate = 0.1
W1 -= learning_rate * dW1
b1 -= learning_rate * db1
W2 -= learning_rate * dW2
b2 -= learning_rate * db2
W3 -= learning_rate * dW3
b3 -= learning_rate * db3
```

```python
print("\nUpdated Weights and Biases:")
print(f"W1: {W1}")
print(f"W2: {W2}")
print(f"W3: {W3}")
print(f"b1: {b1}")
print(f"b2: {b2}")
print(f"b3: {b3}")
```

```

Updated Weights and Biases:
W1: [[-0.34554259  0.4935403  -0.44262008]
 [-0.7502685  -0.54508613 -0.71749508]
 [-0.28692567 -0.3123435  -0.78227928]
 [ 0.8926412  -0.22996956 -0.35290302]]
W2: [[ 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]]
W3: [[-0.05558177 -0.71627873  0.48484404 -0.79736761]]
b1: [[ 0.48137072]
 [-0.07279767]
 [ 0.43346154]
 [ 0.05973635]]
b2: [[0.3552724 ]
 [0.64854159]
 [0.41713561]
 [0.34912214]]
b3: [[1.07507519]]
```

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.

```python
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
```

Otras clases usadas

```python
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()]
```

```python
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()]
```

- La clase Neuron representa una sola neurona en una red neuronal.
- La clase Layer representa una capa de varias neuronas.
- La clase MLP (Multi-Layer Perceptron) representa una red neuronal compuesta por varias capas.

Red Neuronal MultiCapa

```
[ ] n = MLP(3, [4, 4, 1])

W: [Value(data=-0.34108091502837, grad=0.0), Value(data=0.4875444774924518, grad=0.0), Value(data=-0.5786687891714137, grad=0.0)]
b: Value(data=0.42066411352123567, grad=0.0)
W: [Value(data=-0.7406084700295616, grad=0.0), Value(data=-0.5419469493214886, grad=0.0), Value(data=-0.7157868250555495, grad=0.0)]
b: Value(data=-0.06842417572242421, grad=0.0)
W: [Value(data=-0.4315505068114813, grad=0.0), Value(data=-0.3888294993216346, grad=0.0), Value(data=-0.7260616380268912, grad=0.0)]
b: Value(data=0.3917250492343052, grad=0.0)
W: [Value(data=0.8544519713956533, grad=0.0), Value(data=-0.24436625968891335, grad=0.0), Value(data=-0.29745436944045234, grad=0.0)]
b: Value(data=0.07861738679252617, grad=0.0)
W: [Value(data=0.8240668394757076, grad=0.0), Value(data=-0.7925535106179675, grad=0.0), Value(data=0.856718957818575, grad=0.0), Value(data=0.38940887530348656,
b: Value(data=0.2807627460701372, grad=0.0)
W: [Value(data=-0.8401491947426882, grad=0.0), Value(data=0.6005090770290116, grad=0.0), Value(data=-0.710761829952594, grad=0.0), Value(data=-0.5376371931922113
b: Value(data=0.6705844066805486, grad=0.0)
W: [Value(data=-0.9617475006414558, grad=0.0), Value(data=-0.1484109682541901, grad=0.0), Value(data=0.315131084013728, grad=0.0), Value(data=0.843811249537862,
b: Value(data=0.2750476238785533, grad=0.0)
W: [Value(data=0.7589476929906058, grad=0.0), Value(data=-0.6751635852663345, grad=0.0), Value(data=-0.05422983414165872, grad=0.0), Value(data=0.430008053438502
b: Value(data=0.20918398742213418, grad=0.0)
W: [Value(data=-0.3547048938594579, grad=0.0), Value(data=-0.3011280508308256, grad=0.0), Value(data=0.5691627652457749, grad=0.0), Value(data=-0.966577207820160
b: Value(data=0.9979425023631463, grad=0.0)
```

Resultados

```
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: 0.892641203241102
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
```

La clase MLP define una red neuronal usando torch.nn.Module, que es la base para construir modelos en PyTorch.

- La primera capa toma una entrada de tamaño 3 y produce una salida de tamaño 4.

- La segunda capa toma una entrada de tamaño 4 y produce otra salida de tamaño 4.

- La capa de salida toma una entrada de tamaño 4 y produce una única salida.

```python
class MLP(nn.Module):
    def __init__(self, W1, b1, W2, b2, W3, b3):
        super(MLP, self).__init__()
        self.layer1 = nn.Linear(3, 4)
        self.layer2 = nn.Linear(4, 4)
        self.output_layer = nn.Linear(4, 1)

        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
```

Resultados

```python
print("\nPesos y bias después de la retropropagación:")
for name, param in model.named_parameters():
    print(f"{name}: {param.data}")

print(f"\nPérdida final: {loss}")
```

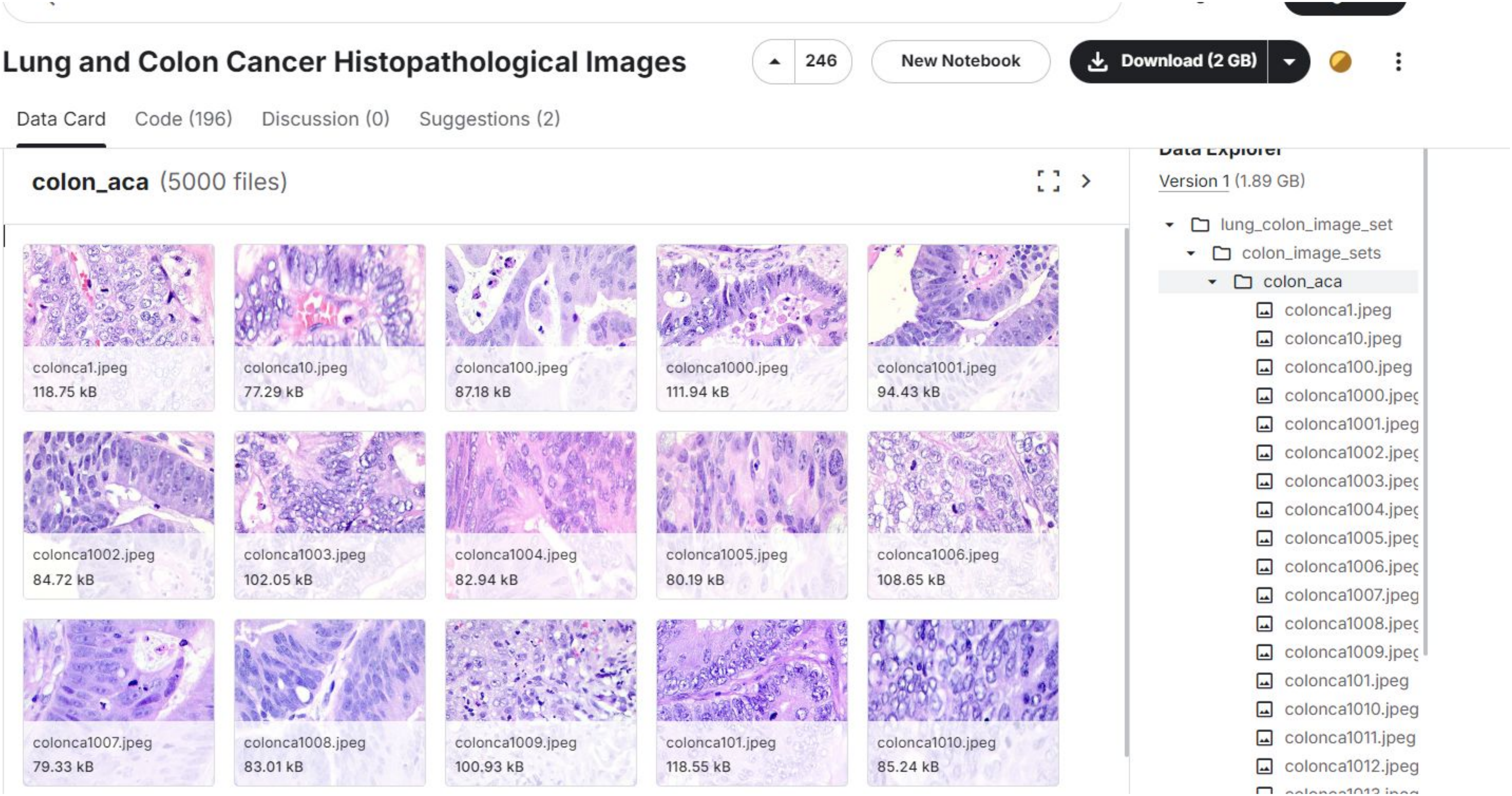
```
Pesos y bias después de la retropropagación:
layer1.weight: tensor([[-0.3652,  0.4924, -0.5307],
        [-0.7427, -0.5432, -0.7167],
        [-0.4137, -0.3822, -0.7352],
        [ 0.8516, -0.2657, -0.3195]], device='cuda:0')
layer1.bias: tensor([[ 0.4201],
        [-0.0663],
        [ 0.3961],
        [ 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
```

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’].

Lung and Colon Cancer Histopathological Image Dataset (LC25000)

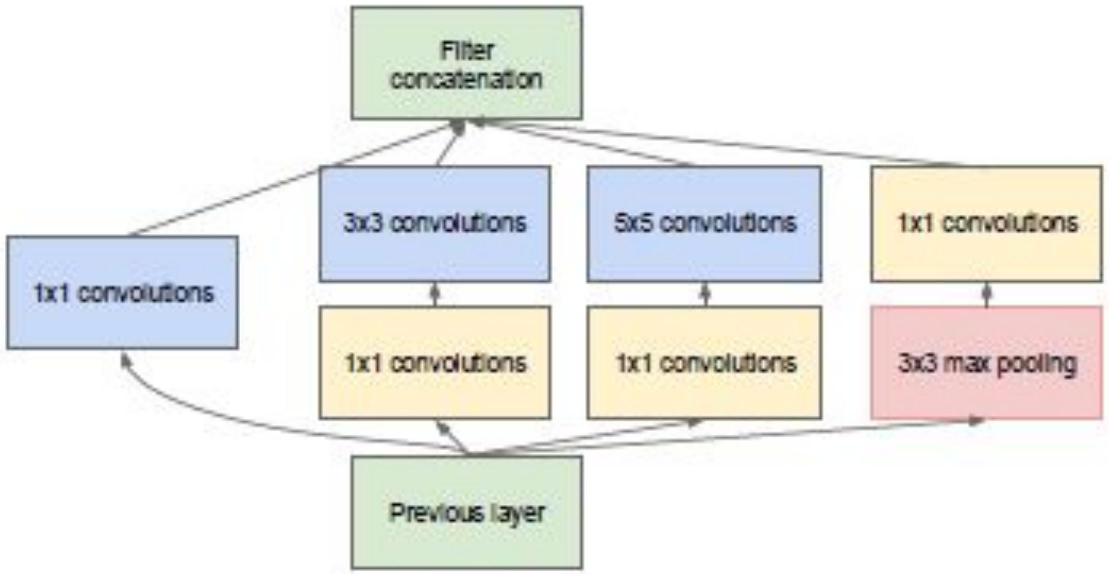
Andrew A. Borkowski, MD*[1,2], Marilyn M. Bui, MD, PhD[2,3], L. Brannon Thomas, MD, PhD[1,2], Catherine P. Wilson, MT[1], Lauren A. DeLand, RN[1], Stephen M. Mastorides, MD[1,2]

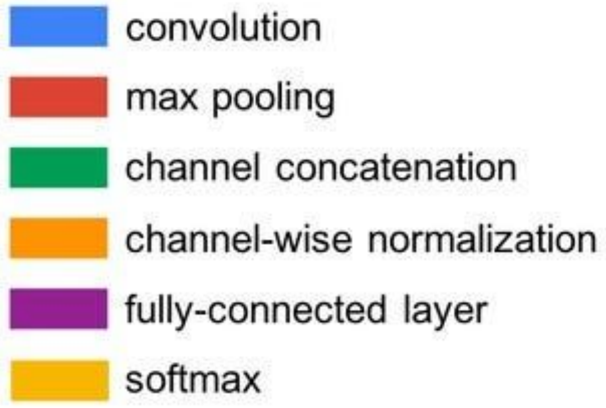
El conjunto de datos contiene 15000 imágenes en color, divididas en 3 clases de 5000 imágenes cada una. Todas las imágenes tienen un tamaño de 768 x 768 píxeles y están en formato jpeg. Las clases son adenocarcinomas de pulmón, carcinomas de células escamosas de pulmón y tejidos pulmonares benignos.

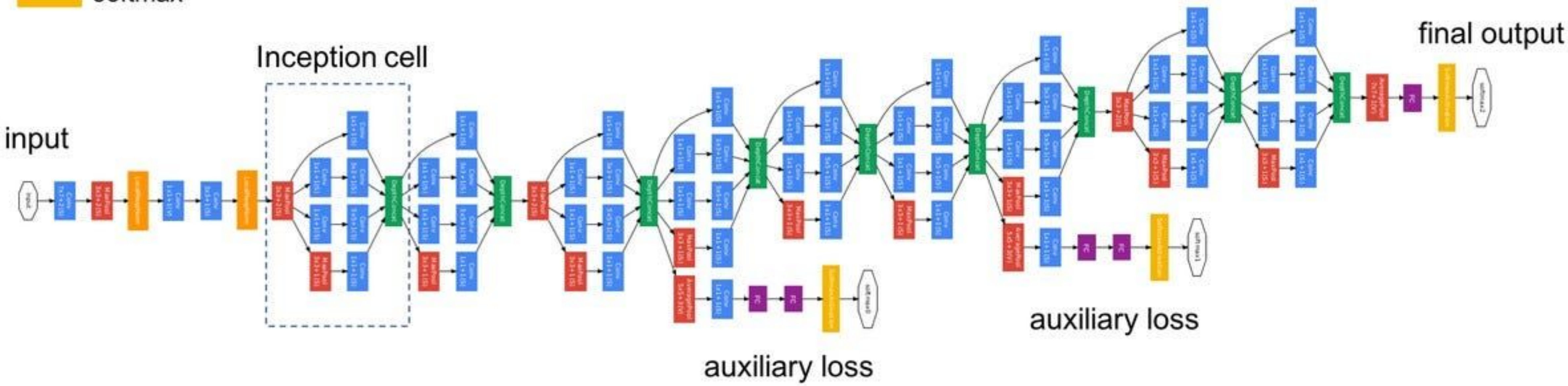

Lung and Colon Cancer Histopathological Images
246
New Notebook
Download (2 GB)
Data Card
Code (196)
Discussion (0)
Suggestions (2)
colon_aca (5000 files)
colonca1.jpeg
118.75 kB
colonca10.jpeg
77.29 kB
colonca100.jpeg
87.18 kB
colonca1000.jpeg
111.94 kB
colonca1001.jpeg
94.43 kB
colonca1002.jpeg
84.72 kB
colonca1003.jpeg
102.05 kB
colonca1004.jpeg
82.94 kB
colonca1005.jpeg
80.19 kB
colonca1006.jpeg
108.65 kB
colonca1007.jpeg
79.33 kB
colonca1008.jpeg
83.01 kB
colonca1009.jpeg
100.93 kB
colonca101.jpeg
118.55 kB
colonca1010.jpeg
85.24 kB
Version 1 (1.89 GB)
lung_colon_image_set
colon_image_sets
colon_aca
colonca1.jpeg
colonca10.jpeg
colonca100.jpeg
colonca1000.jpeg
colonca1001.jpeg
colonca1002.jpeg
colonca1003.jpeg
colonca1004.jpeg
colonca1005.jpeg
colonca1006.jpeg
colonca1007.jpeg
colonca1008.jpeg
colonca1009.jpeg
colonca101.jpeg
colonca1010.jpeg
colonca1011.jpeg
colonca1012.jpeg

Inceptionv1

Módulo Inception

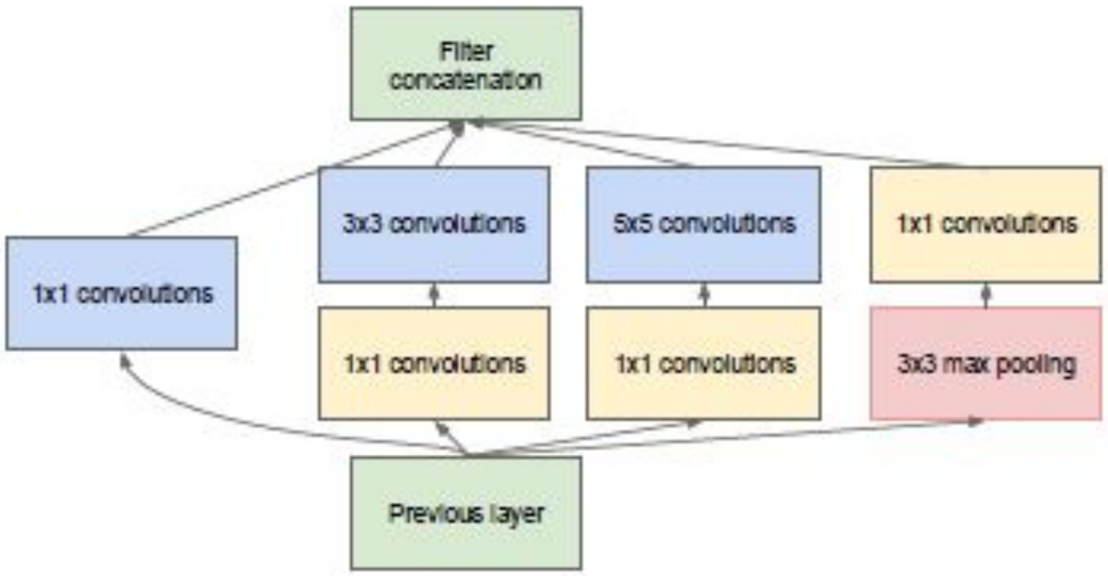






Inceptionv1

Módulo Inception




```python
class Inception_block(nn.Module):
    def __init__(self, in_channels, out_1x1, red_3x3, out_3x3, red_5x5, out_5x5, out_1x1pool):
        super(Inception_block, self).__init__()

        self.branch1 = conv_block(in_channels, out_1x1, kernel_size=1)

        self.branch2 = nn.Sequential(
            conv_block(in_channels, red_3x3, kernel_size=1),
            conv_block(red_3x3, out_3x3, kernel_size=3, stride=1, padding=1)
        )

        self.branch3 = nn.Sequential(
            conv_block(in_channels, red_5x5, kernel_size=1),
            conv_block(red_5x5, out_5x5, kernel_size=5, stride=1, padding=2)
        )

        self.branch4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1),
            conv_block(in_channels, out_1x1pool, kernel_size=1)
        )

    def forward(self, x):
        return torch.cat([self.branch1(x), self.branch2(x), self.branch3(x), self.branch4(x)], 1)
```

Inceptionv1

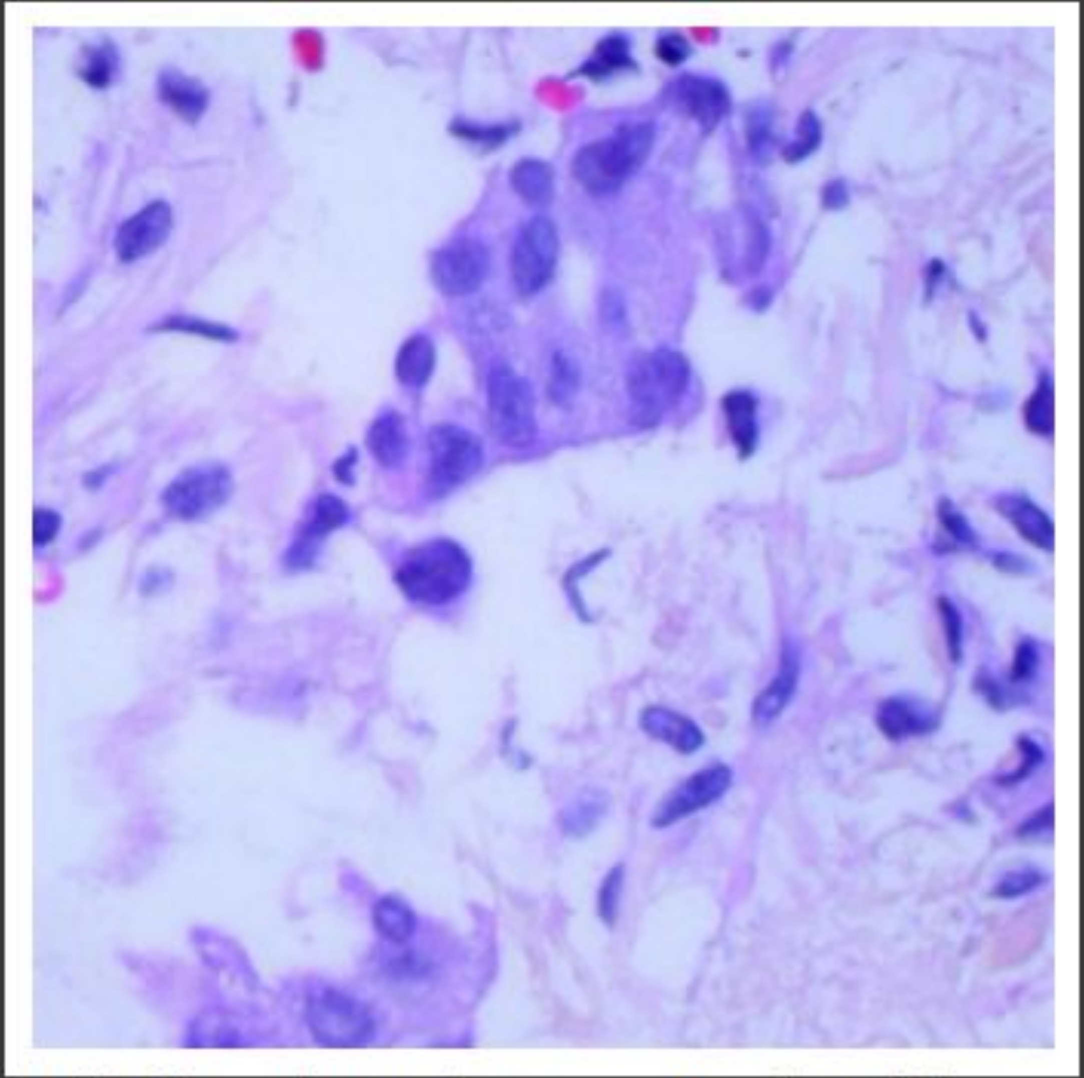
```python
class GoogleNet(nn.Module):
    def __init__(self, in_channels=3, num_classes=1000):
        super(GoogleNet, self).__init__()

        self.conv1 = conv_block(in_channels=in_channels, out_channels=64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
        self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.conv2 = conv_block(in_channels=64, out_channels=192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        self.maxpool2 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.inception3a = Inception_block(192, 64, 96, 128, 16, 32, 32)
        self.inception3b = Inception_block(256, 128, 128, 192, 32, 96, 64)
        self.maxpool3 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.inception4a = Inception_block(480, 192, 96, 208, 16, 48, 64)
        self.inception4b = Inception_block(512, 160, 112, 224, 24, 64, 64)
        self.inception4c = Inception_block(512, 128, 128, 256, 24, 64, 64)
        self.inception4d = Inception_block(512, 112, 144, 288, 32, 64, 64)
        self.inception4e = Inception_block(528, 256, 160, 320, 32, 128, 128)
        self.maxpool4 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.inception5a = Inception_block(832, 256, 160, 320, 32, 128, 128)
        self.inception5b = Inception_block(832, 384, 192, 384, 48, 128, 128)

        self.avpool = nn.AvgPool2d(kernel_size=16)
        self.dropout = nn.Dropout(p=0.4)
        self.fc1 = nn.Linear(1024, num_classes)
```

```python
def forward(self, x):
    x = self.conv1(x)
    x = self.maxpool1(x)
    x = self.conv2(x)
    x = self.maxpool2(x)

    x = self.inception3a(x)
    x = self.inception3b(x)
    x = self.maxpool3(x)

    x = self.inception4a(x)
    x = self.inception4b(x)
    x = self.inception4c(x)
    x = self.inception4d(x)
    x = self.inception4e(x)
    x = self.maxpool4(x)

    x = self.inception5a(x)
    x = self.inception5b(x)
    x = self.avpool(x)

    x = x.reshape(x.shape[0], -1)
    x = self.dropout(x)
    x = self.fc1(x)

    return x
```

Inceptionv1

```
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%
```

Epoch: 10
Accuracy: 97.69%

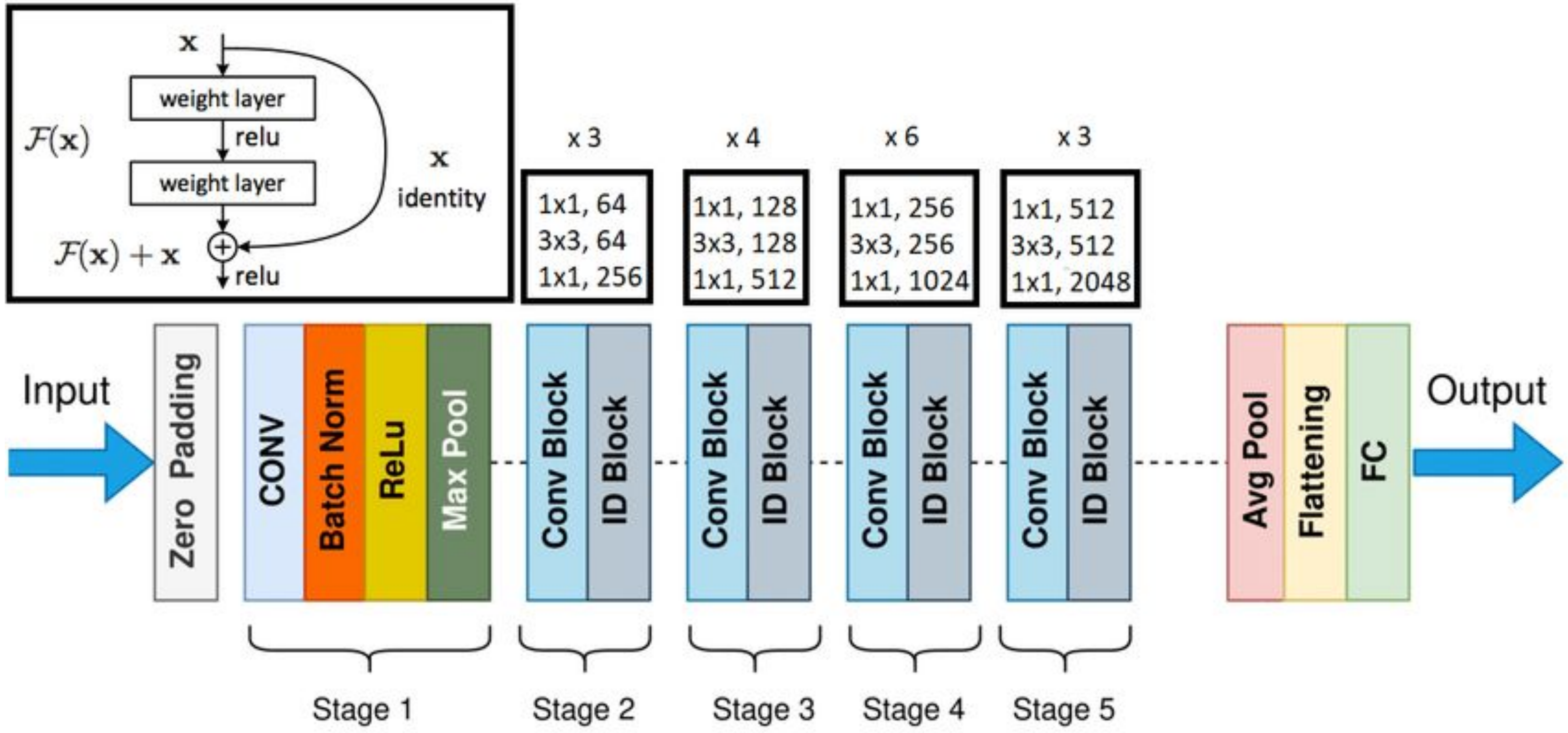


Predicción: adenocarcinoma con 97.00% confidence

ResNet-50

Previene el degradación de la gradiente en el backpropagation

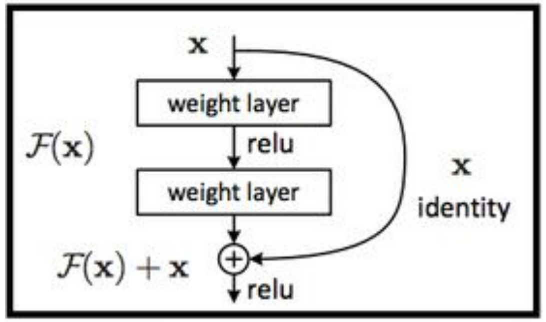
$$H(x) = F(x) + x \qquad \frac{\partial H(x)}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

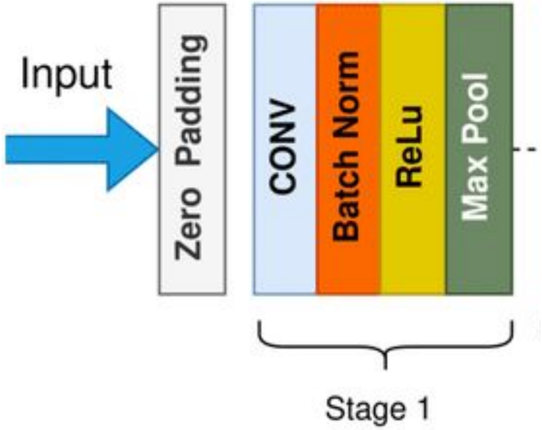


50-layer
7×7, 64, stride 2
3×3 max pool, stride 2

layer name	output size	50-layer
conv1	112×112	7×7, 64, stride 2
conv2_x	56×56	3×3 max pool, stride $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$
conv4_x	14×14	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$
conv5_x	7×7	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	1×1	erage pool, 1000-d fc, s
FLOPs		3.8×10^9

ResNet-50

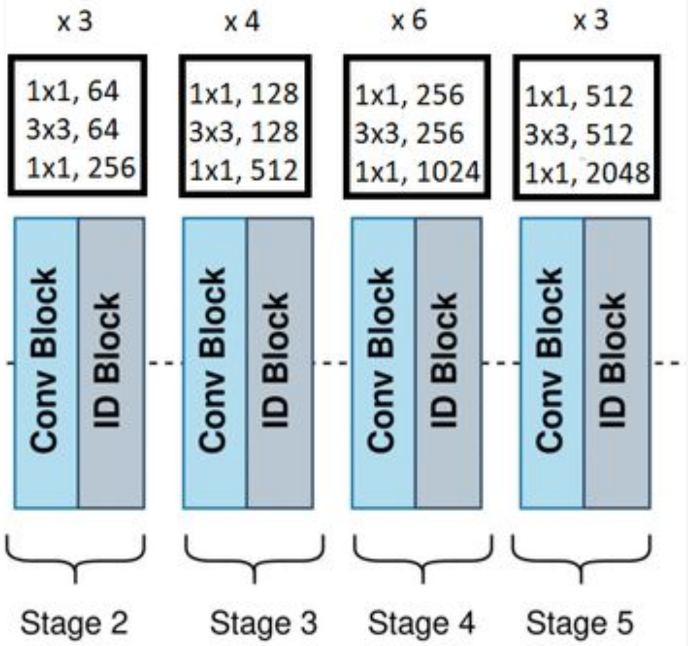


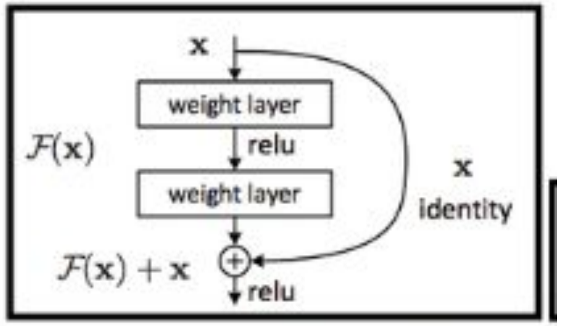


50-layer
7×7, 64, stride 2
3×3 max pool, stride 2

```python
class ResNet50(nn.Module):
    def __init__(self, block, layers, num_classes=3):
        super(ResNet50, self).__init__()
        self.in_channels = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)  # First layer with large 7x7 kernel
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * 4, num_classes)
```

ResNet-50

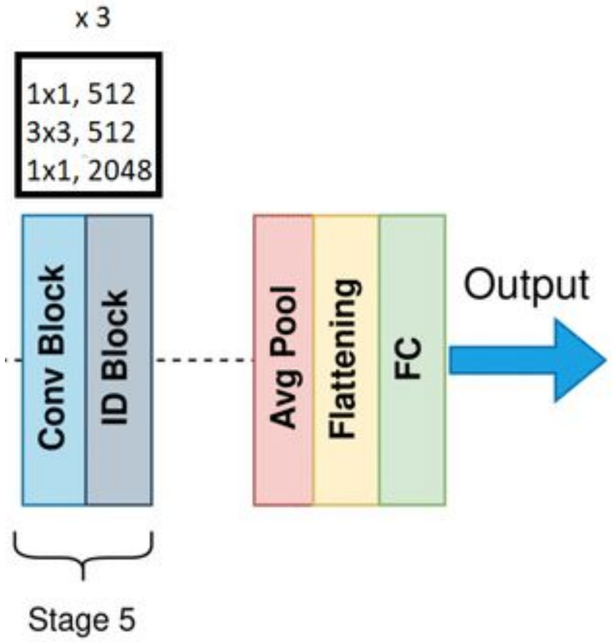





```python
class Bottleneck(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=1, bias=False)  # 1x1 convolution
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)  # 3x3 convolution
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.conv3 = nn.Conv2d(out_channels, out_channels * 4, kernel_size=1, bias=False)  # 1x1 convolution to increase channels
        self.bn3 = nn.BatchNorm2d(out_channels * 4)
        self.relu = nn.ReLU(inplace=True)

        self.downsample = downsample

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

```python
class ResNet50(nn.Module):
    def __init__(self, block, layers, num_classes=3):
        super(ResNet50, self).__init__()
        self.in_channels = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * 4, num_classes)
```

ResNet-50



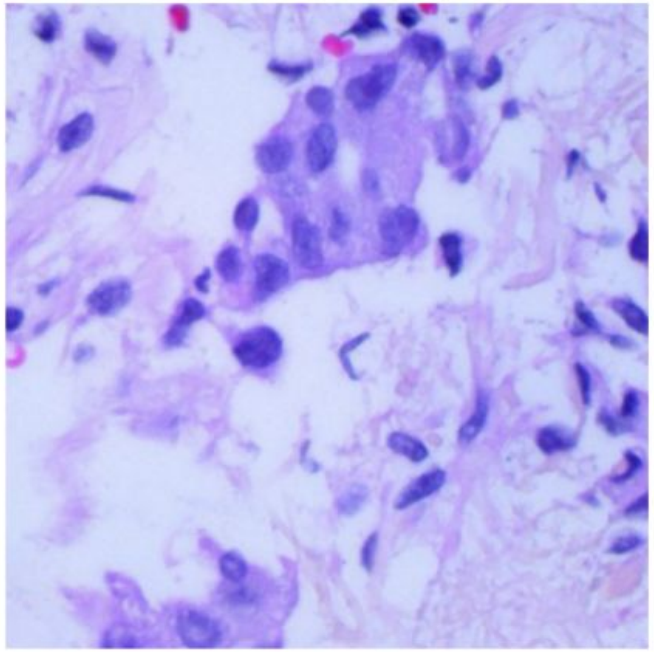

```python
class ResNet50(nn.Module):
    def __init__(self, block, layers, num_classes=3):
        super(ResNet50, self).__init__()
        self.in_channels = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)  # First layer with large 7x7 kernel
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * 4, num_classes)
```

```python
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)  # Flatten the output
        x = self.fc(x)

        return x
```

ResNet-50

```
/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%
```

```python
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')
```

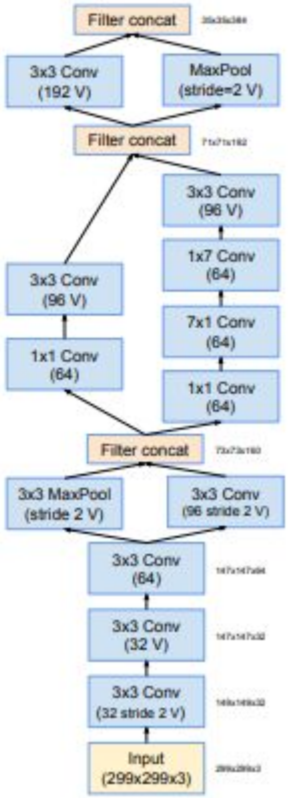


```
Predicción: adenocarcinoma con 98.48% confidence
```

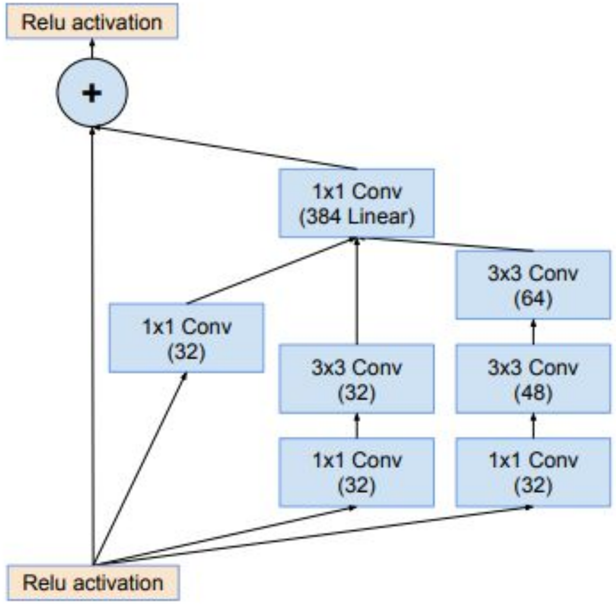
Epoch: 10
Accuracy: 98%

Inception + Resnetv2

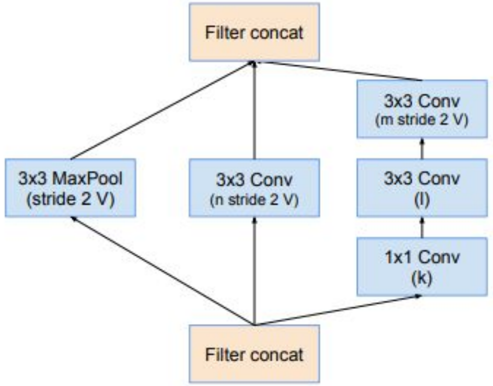
STEM



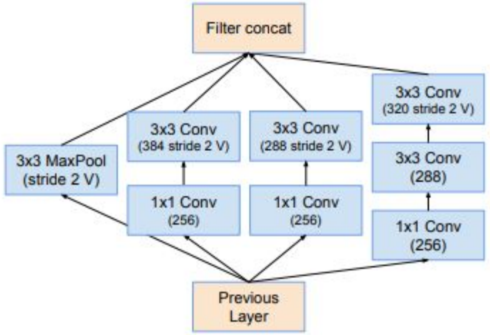
Inception-Resnet A

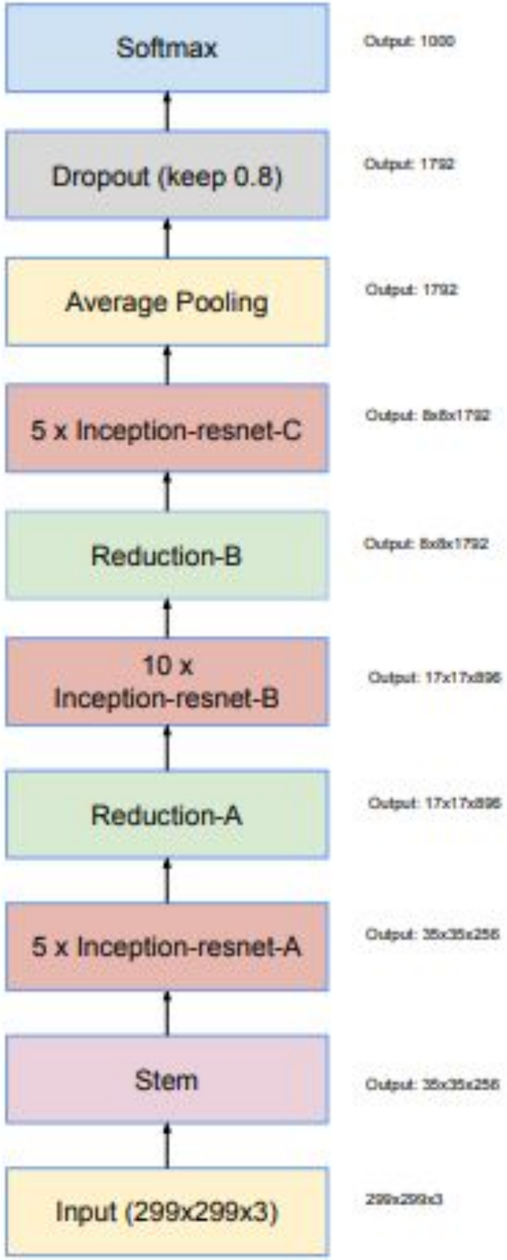


Reduction A



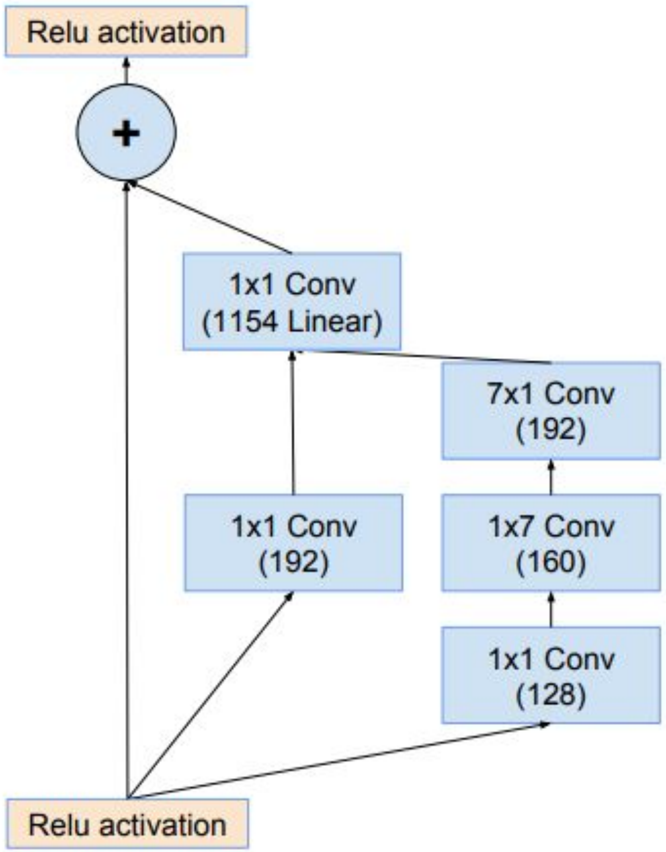
Reduction B



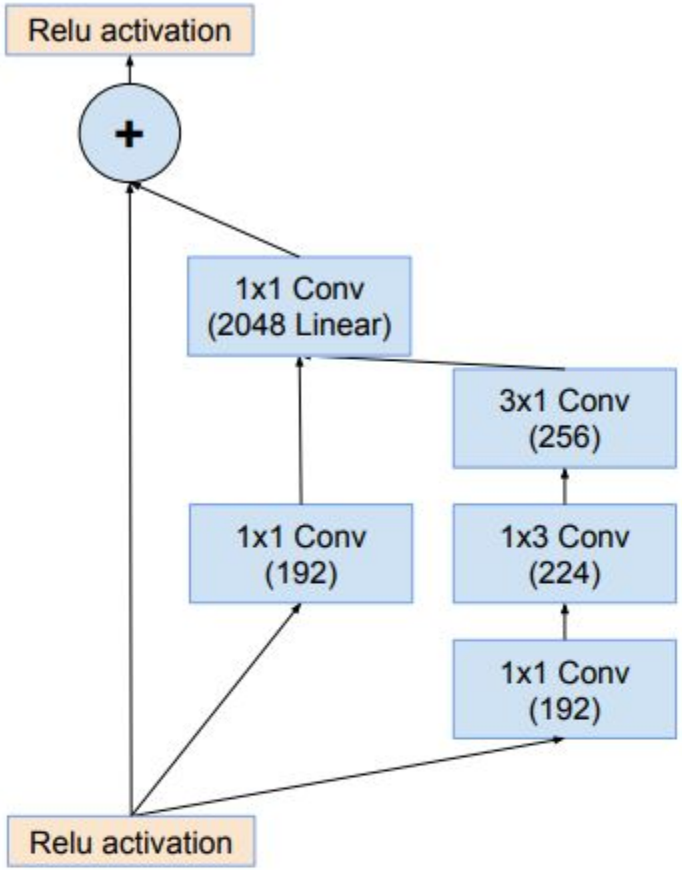


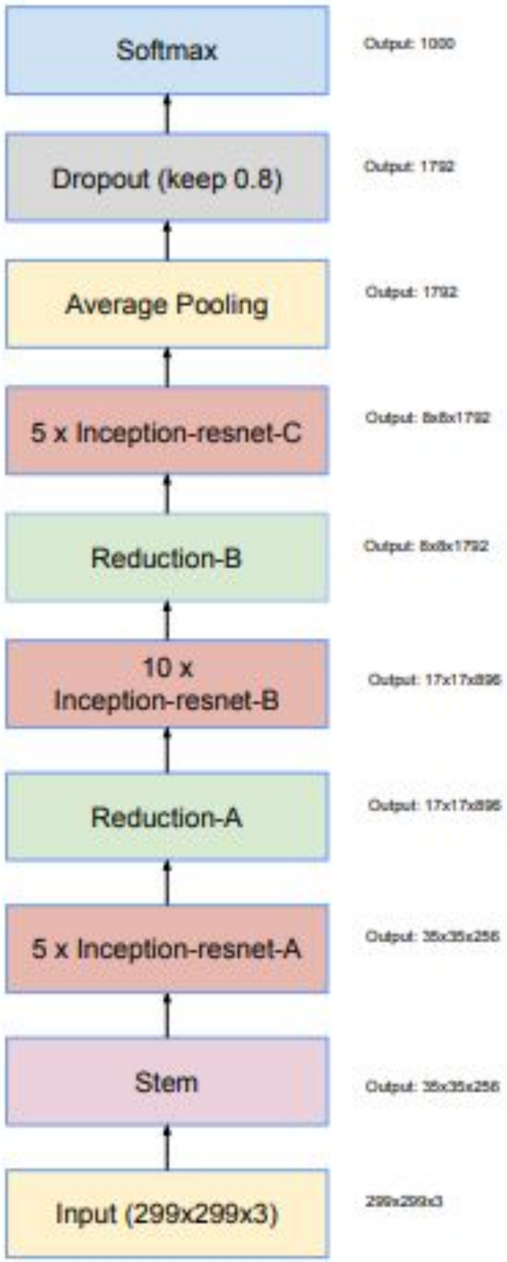
Inception + Resnetv2

Inception-Resnet B



Inception-Resnet C





Inception + Resnetv2

```python
class Stem(nn.Module):
    def __init__(self, in_channels):
        super(Stem, self).__init__()
        self.features = nn.Sequential(
            Conv2d(in_channels, 32, 3, stride=2, padding=0, bias=False), # 149 x 149 x 32
            Conv2d(32, 32, 3, stride=1, padding=0, bias=False), # 147 x 147 x 32
            Conv2d(32, 64, 3, stride=1, padding=1, bias=False), # 147 x 147 x 64
            nn.MaxPool2d(3, stride=2, padding=0), # 73 x 73 x 64
            Conv2d(64, 80, 1, stride=1, padding=0, bias=False), # 73 x 73 x 80
            Conv2d(80, 192, 3, stride=1, padding=0, bias=False), # 71 x 71 x 192
            nn.MaxPool2d(3, stride=2, padding=0), # 35 x 35 x 192
        )
        self.branch_0 = Conv2d(192, 96, 1, stride=1, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(192, 48, 1, stride=1, padding=0, bias=False),
            Conv2d(48, 64, 5, stride=1, padding=2, bias=False),
        )
        self.branch_2 = nn.Sequential(
            Conv2d(192, 64, 1, stride=1, padding=0, bias=False),
            Conv2d(64, 96, 3, stride=1, padding=1, bias=False),
            Conv2d(96, 96, 3, stride=1, padding=1, bias=False),
        )
        self.branch_3 = nn.Sequential(
            nn.AvgPool2d(3, stride=1, padding=1, count_include_pad=False),
            Conv2d(192, 64, 1, stride=1, padding=0, bias=False)
        )
    def forward(self, x):
        x = self.features(x)
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x2 = self.branch_2(x)
        x3 = self.branch_3(x)
        return torch.cat((x0, x1, x2, x3), dim=1)
```

```python
class Inception_ResNet_A(nn.Module):
    def __init__(self, in_channels, scale=1.0):
        super(Inception_ResNet_A, self).__init__()
        self.scale = scale
        self.branch_0 = Conv2d(in_channels, 32, 1, stride=1, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, 32, 1, stride=1, padding=0, bias=False),
            Conv2d(32, 32, 3, stride=1, padding=1, bias=False)
        )
        self.branch_2 = nn.Sequential(
            Conv2d(in_channels, 32, 1, stride=1, padding=0, bias=False),
            Conv2d(32, 48, 3, stride=1, padding=1, bias=False),
            Conv2d(48, 64, 3, stride=1, padding=1, bias=False)
        )
        self.conv = nn.Conv2d(128, 320, 1, stride=1, padding=0, bias=True)
        self.relu = nn.ReLU(inplace=True)
    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x2 = self.branch_2(x)
        x_res = torch.cat((x0, x1, x2), dim=1)
        x_res = self.conv(x_res)
        return self.relu(x + self.scale * x_res)
```

Inception + Resnetv2

```
class Inception_ResNet_B(nn.Module):
    def __init__(self, in_channels, scale=1.0):
        super(Inception_ResNet_B, self).__init__()
        self.scale = scale
        self.branch_0 = Conv2d(in_channels, 192, 1, stride=1, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, 128, 1, stride=1, padding=0, bias=False),
            Conv2d(128, 160, (1, 7), stride=1, padding=(0, 3), bias=False),
            Conv2d(160, 192, (7, 1), stride=1, padding=(3, 0), bias=False)
        )
        self.conv = nn.Conv2d(384, 1088, 1, stride=1, padding=0, bias=True)
        self.relu = nn.ReLU(inplace=True)
    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x_res = torch.cat((x0, x1), dim=1)
        x_res = self.conv(x_res)
        return self.relu(x + self.scale * x_res)
```

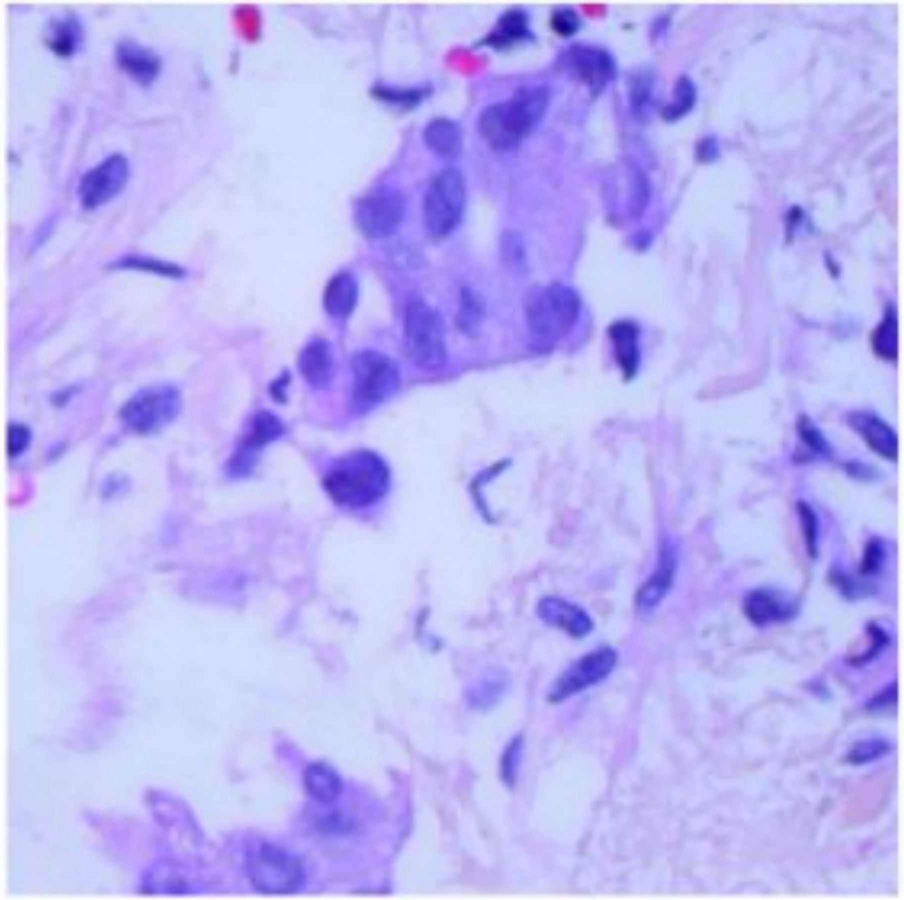
```
class Inception_ResNet_C(nn.Module):
    def __init__(self, in_channels, scale=1.0, activation=True):
        super(Inception_ResNet_C, self).__init__()
        self.scale = scale
        self.activation = activation
        self.branch_0 = Conv2d(in_channels, 192, 1, stride=1, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, 192, 1, stride=1, padding=0, bias=False),
            Conv2d(192, 224, (1, 3), stride=1, padding=(0, 1), bias=False),
            Conv2d(224, 256, (3, 1), stride=1, padding=(1, 0), bias=False)
        )
        self.conv = nn.Conv2d(448, 2080, 1, stride=1, padding=0, bias=True)
        self.relu = nn.ReLU(inplace=True)
    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x_res = torch.cat((x0, x1), dim=1)
        x_res = self.conv(x_res)
        if self.activation:
            return self.relu(x + self.scale * x_res)
        return x + self.scale * x_res
```

```
class Inception_ResNetv2(nn.Module):
    def __init__(self, in_channels=3, classes=1000, k=256, l=256, m=384, n=384):
        super(Inception_ResNetv2, self).__init__()
        blocks = []
        blocks.append(Stem(in_channels))
        for i in range(10):
            blocks.append(Inception_ResNet_A(320, 0.17))
        blocks.append(Reduction_A(320, k, l, m, n))
        for i in range(20):
            blocks.append(Inception_ResNet_B(1088, 0.10))
        blocks.append(Reduciton_B(1088))
        for i in range(9):
            blocks.append(Inception_ResNet_C(2080, 0.20))
        blocks.append(Inception_ResNet_C(2080, activation=False))
        self.features = nn.Sequential(*blocks)
        self.conv = Conv2d(2080, 1536, 1, stride=1, padding=0, bias=False)
        self.global_average_pooling = nn.AdaptiveAvgPool2d((1, 1))
        self.linear = nn.Linear(1536, classes)

    def forward(self, x):
        x = self.features(x)
        x = self.conv(x)
        x = self.global_average_pooling(x)
        x = x.view(x.size(0), -1)
        x = self.linear(x)
        return x
```

Inception + Resnetv2

```
print(f'Accuracy on test set: {100 * correct / total:.2f}%')

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%
```

Epoch: 10
Accuracy: 96.62%



Predicción: adenocarcinoma con 99.06% confidence

RESUMEN

Arquitectura	Nro 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

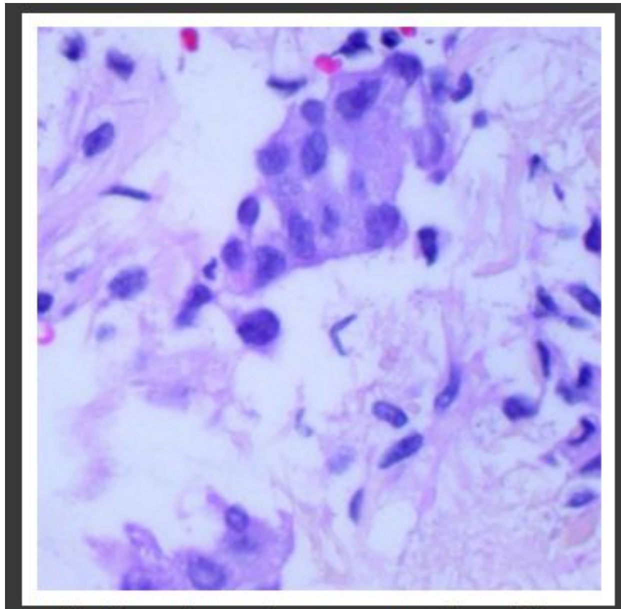
GPU Colab T4 16GB
GPU local: GTX 1060 6GB

Tiempo Google Colab: 2h
Tiempo Local: 5h

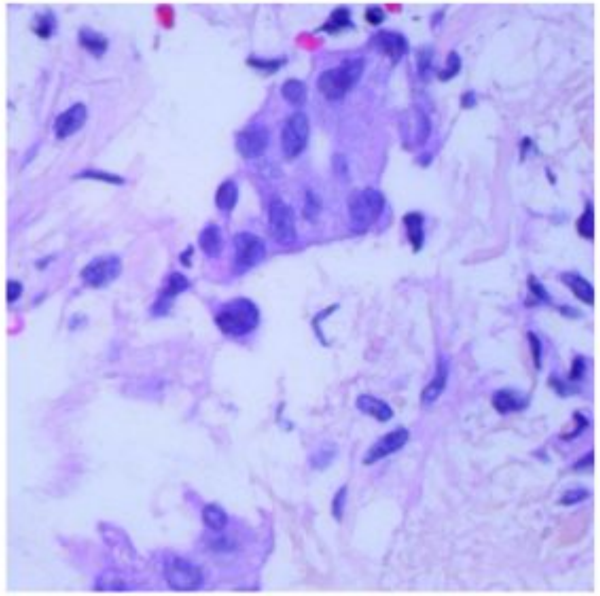
Inceptionv1

Inception + Resnetv2

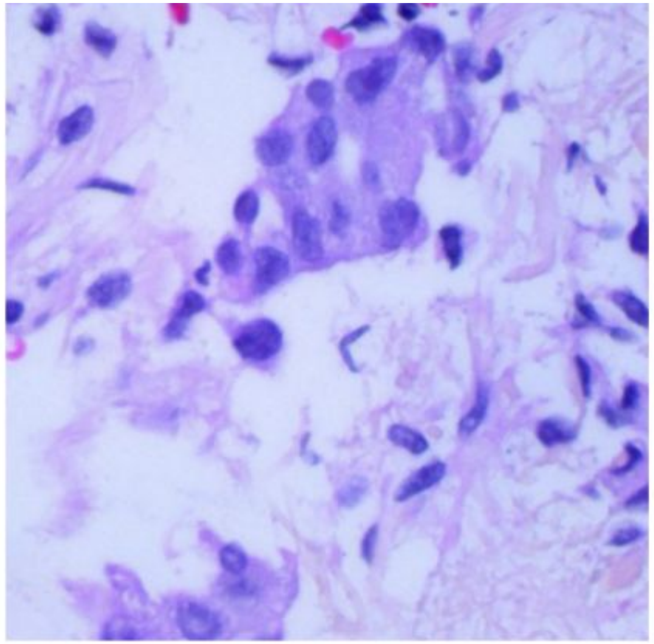
ResNet-50



Predicción: adenocarcinoma con 97.00% confidence



Predicción: adenocarcinoma con 99.06% confidence



Predicción: adenocarcinoma con 98.48% confidence

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

SECC. A

CURSO REDES NEURONALES Y APRENDIZAJE PROFUNDO

GRACIAS