Reconocimiento de letras manuscritas con redes neuronales artificiales

Tercera y cuarta entrega

Edson Raul Cepeda Marquez
Sonia Alejandra Treviño Rivera

1 de junio de 2020

1. Resumen

Este proyecto esta alojado en un repositorio publico, todo código correspondiente fue escrito en ingles para asegurar la mayor accesibilidad. El proyecto consiste en una aplicación web que implementa una red neuronal artificial capaz de reconocer letras manuscritas. La aplicación web en cuestión está desarrollada con tres tecnologías que se conectan y permiten su uso:

1. **Angular**: Un framework de código libre desarollado por Google que facilita el desarrollo de aplicaciones web de una sola página (SPA: Single Page Applicaction). Tiene la ventaja de separar completamente el frontend y el backend lo que permite seguir facilmente el patrón de diseño MVC (Modelo-Vista-Controlador).
2. **Express**: Un framework de desarrollo web que utiliza el entorno de ejecución Node JS y facilita la creación de REST APIs.
3. **Python**: Un lenguaje de programación interpretado orientado a objetos. Tiene la ventaja de ser simple, lo que permite escibir código legible. Tiene una amplia comunidad de desarrollo y una gran cantidad de librerias para todo tipo de tareas.

En la figura 1 se muestra un diagrama que representa el flujo de información entre las tres tecnologías. Se puede observar que la información fluye en ambas direcciones.

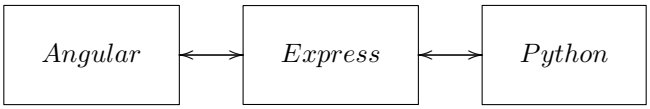

Figura 1: Flujo de información de la aplicación.

La aplicación web además de contener este proyecto, tiene otras secciones asignadas para proyectos futuros. Para este proyecto la sección importante es “Imchar”. Esta sección de la aplicación web contiene una cuadricula con la que se puede interactuar a traves del ratón.

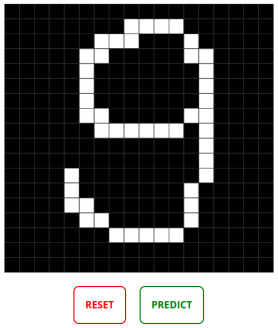

Figura 2: Cuadricula de la aplicación.

Esta es la puerta para el uso de la red neuronal artificial. En esta cuadricula es posible dibujar letras.
Al dibujar una letra en la cuadricula y presionar el botón de "Predeict" la aplicación web realiza una petición al servidor de la REST API de express, la cual recibe el estado de la cuadricula y sus dimensiones.
Posteriormente el servidor de express ejecuta un proceso asincrono de python el cual ejecuta un script que lleva a cabo una transformación de los datos enviados en la petición
Por ultimo los datos transformados de la cuadricula son procesados por la red neuronal previamente entrenada, y devuelve una respuesta al frontend convertida en el caracter que la red neuronal predice (Figura 3).

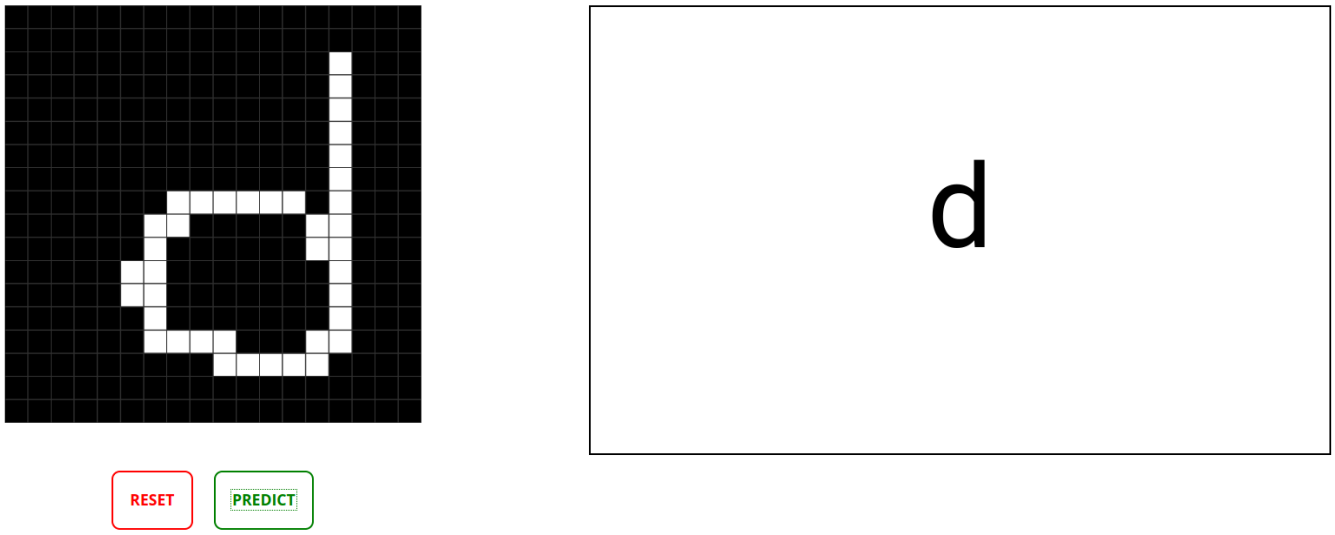

Figura 3: Respuesta de la red neuronal, también incluye el botón "Reset" para borrar la cuadricula.

Para el desarrollo de la red neuronal artificial no se utilizo ninguna librería, todo código correspondiente al algoritmo de aprendizaje fue escrito desde cero.
Así mismo la red neuronal que procesa la información fue entrenada con un conjunto de datos creado desde cero, para esto se implemento una ruta oculta (figura 4) en la aplicación web en la que es posible utilizar la cuadricula para generar un conjunto de entranamiento.

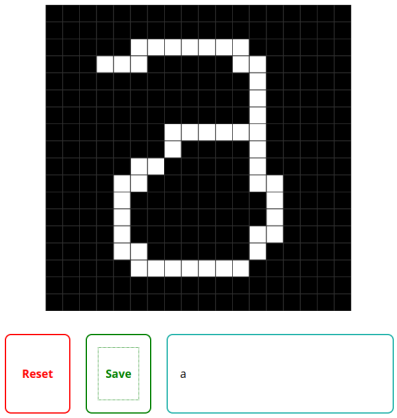

Figura 4: Ruta oculta para la generación de datos de entrenamiento

Los resultados de entrenar la red neuronal con estos datos son que, para las 26 letras del abecedario en ingles por lo menos 21 son predichas correctamente. Las letras que presentan más problemas para ser predichas correctamente son aquellas que comparten cierta similaridad. La red neuronal fue entrenada multiples veces para poder llegar a una configuración de hiperparametros con un mejor rendimiento.

2. Introducción

El problema que se intenta resolver al diseñar este sistema, es el de reconocer texto manuscrito para su digitalización. Se diseña una red neuronal convolucional que es entrenada con un conjunto de datos que contiene pequeñas imagenes de letras del abecedario en ingles. En la sección 1 de este documento, se indica en que consiste el proyecto.
La teoría necesaria para la realización de este proyecto se encuentra en la sección 3
En la sección 4 se revisa la implementación a detalle del sistema al igual que el diseño de la red neuronal. Por utlimo en la sección **??** se incluye una conclusión de lo que fue el proyecto incluyendo una introspección del equipo y trabajo a futuro. Todas la referencias utilizadas para este proyecto se incluyen en la sección de referencias.

3. Marco teórico

Una red neuronal artificial es un modelo matemático y un algoritmo de aprendizage automático inpirado en el funcionamiento de las neuronas biologicas que se encuentran en el cerebro. El elemento básico de una red neuronal es la neurona. Una neurona se compone de cuatro partes fundamentales:

1. **Dendrita**: Son ramificaciónes que conectan con el el cuerpo celular de una neurona, es la parte encargada de transportar los impulsos electricos.
2. **Cuerpo celular**: Es la parte de la neurona encargada de procesar la suma de los impulsos electricos recibidos de las dentritas. Posteriormente, se decide si disparara otra señal electrica de salida por el axon mediante un umbral.
3. **Axon**: Son las ramificaciones finales de la neurona, transportan el impulso de salida de la neurona tras haber sido procesado en el cuerpo celular.
4. **Sinapsis**: El punto de contacto entre el axon de una neurona y la dendrita de otra neurona.

Es la organización de las neuronas y la fuerza de las sinapsis lo que establece la función principal de una neurona. En la figura 5 se puede observar cada una de las partes de la neurona.

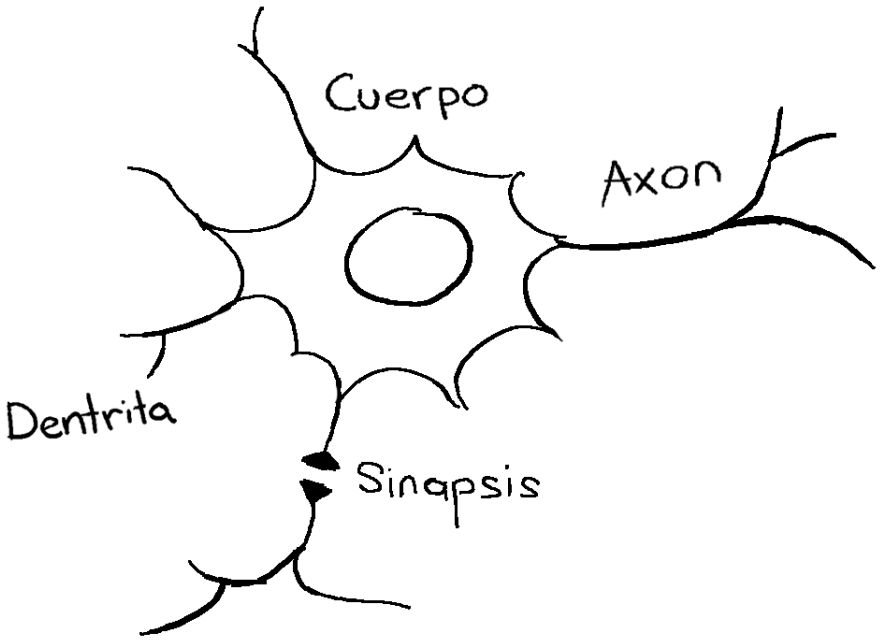

Figura 5: Neurona biologica

Con esto es posible construir un modelo matemático de una neurona artificial (figura 6). En este modelo se tiene un escalar p que corresponde a la señal del impulso electrico recibido de otras neuronas, p es multiplicado por w que corresponde a la fuerza de la sinapsis. Este producto wp es enviado a una sumatoria y se incluye una entrada de sesgo b con un valor de 1. La salida de la sumatoria n va a una función de activación f lo cual produce un escalar de salida a.

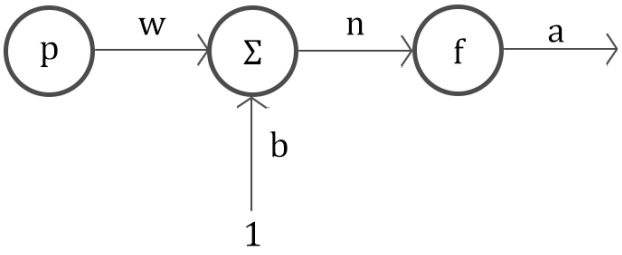

Figura 6: Modelo matemático de neurona artificial.

De esta manera la salida de la neurona es calculada de la forma $a = f(wp + b)$.
La organización de multiples capas y multiples neuronas es lo que le da el poder a este algoritmo. A esto se le conoce como **red neuronal profunda**. Una red neuronal es también conocida como un aproximador de funciones universal, puesto es capaz de aproximar cualquier función teniendo la configuración adecuada.
El modelo matematico de una capa de una red neuronal profunda es, un vector **p** de tamaño R como entrada,un determinado número de neuronas S, seguido de una matriz de pesos **W** con un tamaño $S \times R$ y un vector **b** de sesgo de tamaño $S \times 1$. La operación **Wp + b** es enviada a una sumatoria que resulta en un vector de resultados **n** de tamaño $S \times 1$ y que se dirige a una función de activación f que da una vector **a** de tamaño $S \times 1$ como salida. La salida de una capa es la entrada de la otra. Para definir la arquitectura de una red neuronal se necesita definir una serie de parametros:

1. Número de capas
2. Número de neuronas por capa

3. Tasa de aprendizage
4. Funciones de activación de las neuronas
5. Función de error de la red

A estos se les conoce como **hiperparametros de la red**. Una red neuronal profunda (figura 7) recibe su nombre del hecho que tiene una cantidad considerable de capas intermedias llamadas **capas ocultas**.

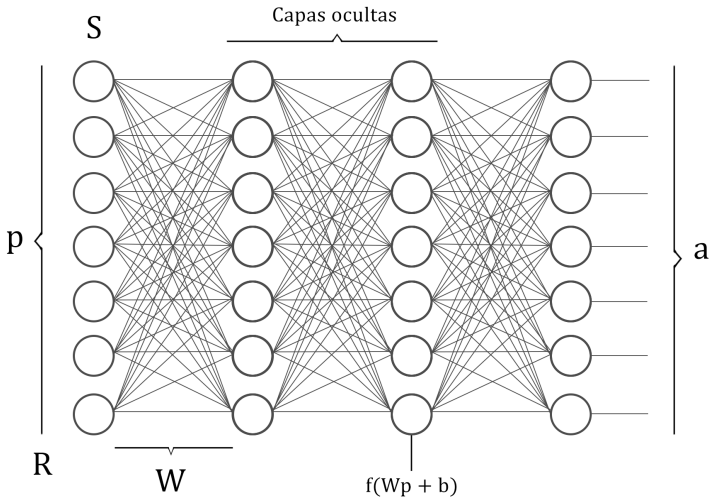

Figura 7: Diagrama representativo de una red neuronal profunda

Al definir el número de neuronas de una capa también se tiene que escoger una función de activación adecuada. Existen distintas funciones de activación que son comúnmente usadas en las redes neuronales. Las tres más comúnes son las siguientes:
Función sigmoidal o logistica: Esta función se encuentra en el rango de 0 a 1. Es muy útil para modelos que requieren predecir una probabilidad.

$$f(x) = \frac{1}{1 + \exp^{-} x} \tag{1}$$

Función tangente hiperbolica: Tambíen es una función logistica pero mejorada, el rango de esta función esta entre -1 a 1. Las ventajas de usar esta función es que penaliza fuertemente las entradas negativas. Es comúnmente usada para la clasificación de dos clases.

$$f(x) = \frac{1 - \exp^{-} 2x}{1 + \exp^{-} 2x} \tag{2}$$

ReLU (Unidad lineal rectificadora): Esta función es la más usada en las capas ocultas de las redes profundas puesto que evita el problema del desvanecimiento del gradiente que presentan las dos funciones anteriores. Esta función tiene un rango de 0 a infinito. Las entradas iguales o menores a cero se hacen cero, por lo que en algunos casos esto impide que se aprenda bien de los datos. Existen modificaciones de esta función que pretenden arreglar este problema como **Leaky ReLU** o **Randomized ReLU**.

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x >= 0 \end{cases} \tag{3}$$

Las tres funciones anteriores son facílmente diferenciables lo que las hace adecuadas para el algoritmo de aprendizaje. El algoritmo más utilizado es el de propagación hacía atras o mejor conocido en ingles como **Backpropagation**.
Backpropagation es un algoritmo de aprendizaje que tiene como objetivo minimizar una función de error mediante el uso del **descenso del gradiente**. El entrenamiento de una red neuronal con este algoritmo consta de tres etapas:

1. **Feedfoward**: Esta etapa consiste en propagar las entradas hacia delante en la red para obtener un resultado, este resultado se compara con el resultado esperado de la entrada y se calcula un error.

2. **Backpropagation**: Esta etapa consiste en calcular cuanto cambia el error si se ajustan cada uno de los pesos de la red neuronal. Se calcula una sensibilidad para los pesos derivando parcialmente la función de error respecto a cada peso.

3. **Actualización de pesos**: Usando las sensibilides calculadas en la etapa anterior se realiza una actualización en los valores de los pesos.

Este procedimiento es repetido multiples veces y con distintas entradas, estas son las **iteraciones** del entrenamiento. De esta manera la red es alimentada con suficientes ejemplos como para generalizar correctamente y reducir el error lo mejor posible. Existen distintas architecturas que resuelven distintos tipos de problemas. Dos de las más comunes son:

1. **Red neuronal convolucional**: es usada comúnmente para la clasificación de imagenes. Esta red neuronal consta de una serie de capas de extracción de caracteristicas que utiliza tecnicas de visión computacional. Las salidas de estas capas son conectadas con una serie de capas finales densas (totalmente conectadas) en las que se lleva a cabo el aprendizage.

2. **Red neuronal recurrente**: es usada ampliamente en el campo de **procesamiento de lenguaje natural**, estas redes neuronal reciben su nombre debido a que existe una retroalimentación entre las capas.

Las redes neuronales como algoritmos de aprendizage automatico supervisado, realizan el proceso de aprendizage con conjuntos de datos previamente etiquetados. La calidad de este conjunto de datos refleja la calida de las predicciones de una red neuronal.
Otro de los factores que influyen en el rendimiento de la red neuronal, es la configuración de los hiperparametros. Existen problemas que requieren más o menos neuronas, otros que requieren más capas o menos capas. Es por esto que los hiperparametros de una red neuronal también pueden ser configurados automaticamente con el uso de otros algoritmos tales como **el algoritmo genetico**.

4. Desarrollo

La técnica seleccionada es una **red neuronal convolucional**. Una red convolucional esta compuesta por una serie de capas con diferente organización, estas capas son:

4.1. Capas de convoluciones

Estas son las capas en la que se lleva a cabo la extracción de caracteristicas de una imagen. Las imagenes pueden ser representadas como matrices tridimensionales para imagenes en colores que corresponderian a cada uno de los canales (R: Rojo, G: Verde, B: Azul) o como matrices bidimensionales que representan imagenes en escala de grises (figura 8).

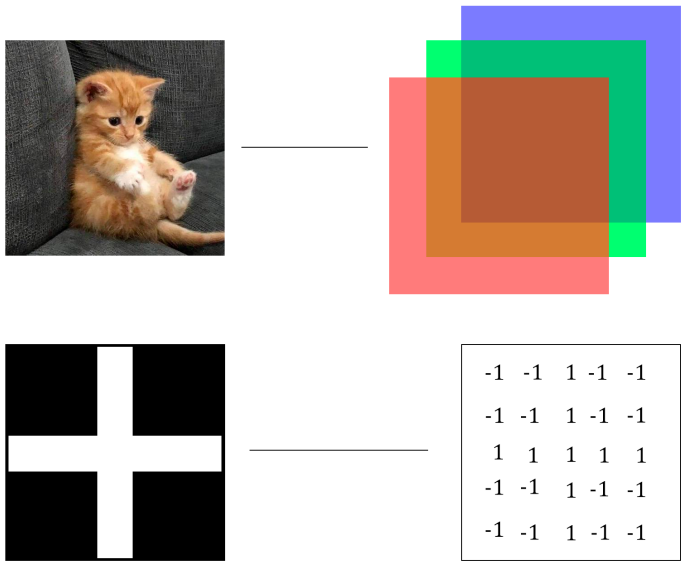

Figura 8: Representación de imagenes en matrices

Una convolución se refiere al proceso de aplicar un filtro a una imagen, los filtros utilizados en estas capas son llamados **kernel**. Ejemplos comunes de kernels se pueden observar en la figura:

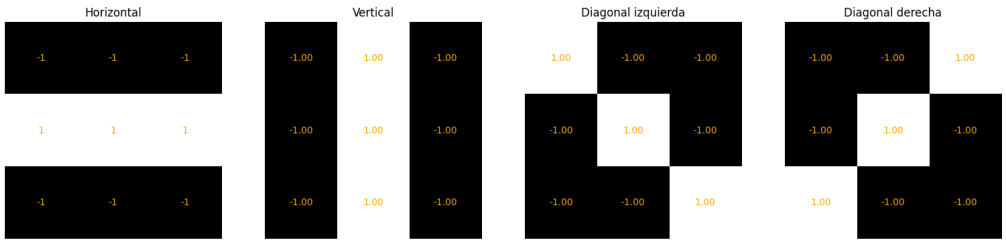

Figura 9: Representación de imagenes en matrices

Estos kernels son pequeñas matrices que se desplazan sobre la imagen realizando un producto punto, el pixel central se cons de manera queerva y se origina una nueva imagen.A esta operación se le llama **correlación cruzada** De esta manera los elementos de la matriz que más encajan con la imagen se veran resaltados en la nueva imagen. Al realizar esta operación de convolución es común que se reduzca la dimensionalidad de la matriz original, por lo que se utiliza la técnica de **padding** que genera un borde de ceros para el pixel central del kernel encaje con el primer pixel de la imagen, de esta manera se conservan las dimensiones de la imagen original. Para la implementación de esta técnica se creo una clase especializada que incluye la lógica del algoritmo de convolución.
Se crea un archivo en el que solo se declara las matrices de los kernels. Se incluye kernels para lineas verticales, lineas horizontales, lineas diagonales, difuminación, contornos, nitidez.

```
import numpy as np

horizontal_line = np.array([[-1,-1,-1],[1,1,1],[-1,-1,-1]])

vertical_line = np.array([[-1,1,-1],[-1,1,-1],[-1,1,-1]])

diagonal_left = np.array([[1,-1,-1],[-1,1,-1],[-1,-1,1]])

diagonal_right = np.array([[-1,-1,1],[-1,1,-1],[1,-1,-1]])

```

```
cross = np.array([[1,-1,1],[-1, 1,-1],[1,-1, 1]])

circular = np.array([[-1, 1, -1],[ 1,-1,  1],[-1, 1, -1]])

sharpening = np.array([[0,-1,0],[-1,5,-1],[0,-1,0]])

blur = np.array([[0.0625,0.125,0.0625],[0.125,0.25,0.125], [0.0625,0.125,0.0625]])

bottom_sobel = np.array([[-1,-2,-1],[0,0,0],[1,2,1]])

emboss = np.array([[-2,-1,0],[-1,1,1],[0,1,2]])

outline = np.array([[-1,-1,-1],[-1,8,-1],[-1,-1,-1]])

top_sobel = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])
```

En un archivo separado se importan los kernels y se escribe la función que realiza la operación de correlación cruzada. Se empieza por definir los tamaños de las entradas para poder recorrer ambas matrices, la de kernel con un tamaño $h \times w$ y la de entrada $H \times W$. La función puede ser ejecutada con un parametro opcional que es el del padding, en caso de que sea una valor verdadero se crea una matriz expandida con zeros en los borders. La matriz origina con un tamaño de $h \times w$ se ve expandida a una matriz de tamaño $(h+2) \times (w+2)$, esto debido a que se agrega una fila de ceros por encima, por debajo, por la izquieda y por la derecha. En caso de que el valor de padding sea falso, no se realiza la expansión y directamente se hacen las operaciones de correlación. De esta manera la nueva matriz que representa la imagen termina con un tamaño de $(H-h+expansion+1) \times (W-w+expansion+1)$. El valor de retorno de la función es la nueva matriz.

```
import numpy as np
import Kernels

def crossCorr2d(input, kernel,padding = False):
    h,w = kernel.shape #Dimensions of kernels
    H,W = input.shape #Dimensions of input matrix
    pad = 0
    expansion = 0
    if(padding):
        pad = 1
        expansion = pad * 2 #Two borders of zeros vertical and horizontal
        paddedInput = np.zeros((H+expansion, W+expansion))
        #Assigning old values in the center of the matrix with padding
        for i in range(input.shape[0]):
            for j in range(input.shape[1]):
                paddedInput[i+pad,j+pad] = input[i,j]
        #Replace the original input with the padded input matrix
        input = paddedInput
    output = np.zeros((H-h+expansion+1, W-w+expansion+1)) #Initializing new matrix
    for i in range(output.shape[0]):
        for j in range(output.shape[1]):
            output[i,j] = (input[i:i+h, j:j+w] * kernel).sum()/(h*w) #Asigning cross correlation
    operations
    return output
```

Para comprobar el funcionamiento de la función, se exploran los resultados de aplicar seis kernels distintos a una misma imagen en escala de grises.
Se hace uso de la librería matplotlib para la visualización de los resultdos. En la figura se puede observar como cambian los resultados con distitos kernels y cuales son las caracteristicas de la imagen que más se ven resaltadas con cada uno.

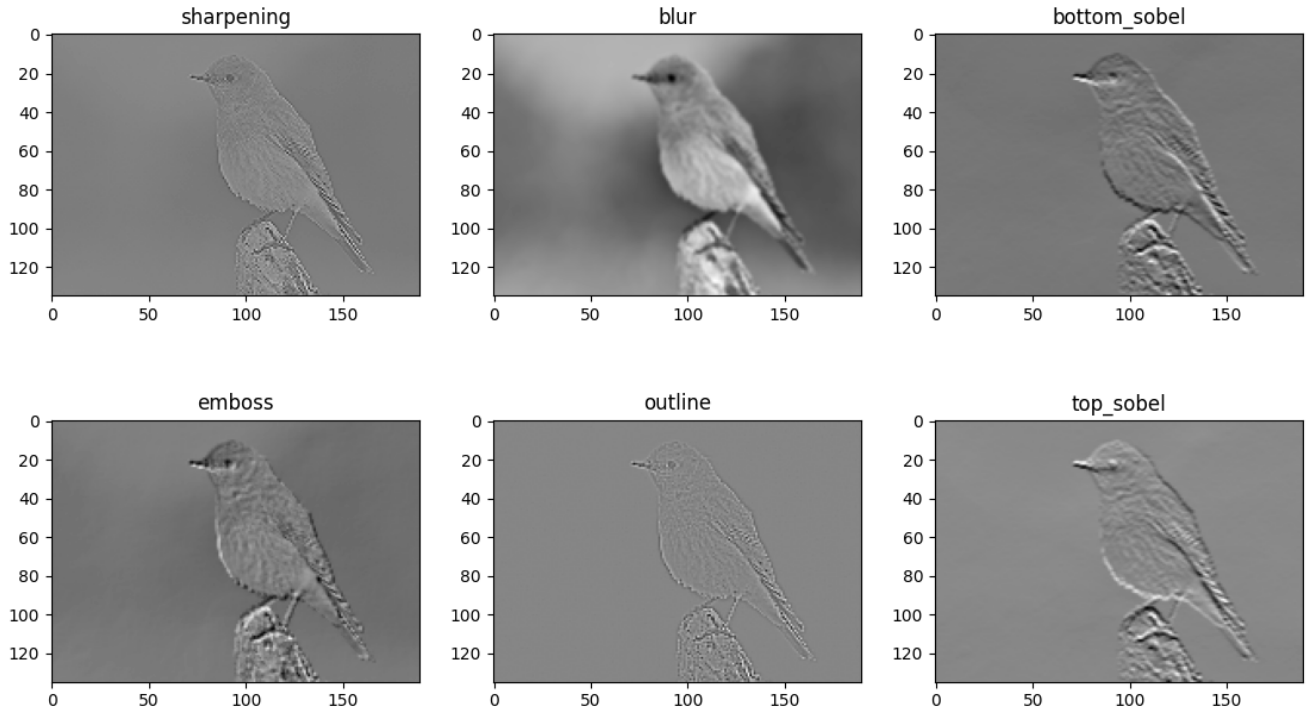

Figura 10: Resultado de aplicar los kernels a la imagen

4.2. Capas de pooling

La operación de pooling realiza una reducción de dimensionalidad en una matriz, concretamente la técnica de **max pooling** funciona de la siguiente manera:
Se utiliza el concepto de una ventana con un tamaño determinado, que se desplaza en el interior de la matriz. La ventana se define por su tamaño y por los pasos que avanza. Se toma el valor máximo de la ventana y se asigna a la nueva matriz reducida (figura 11).

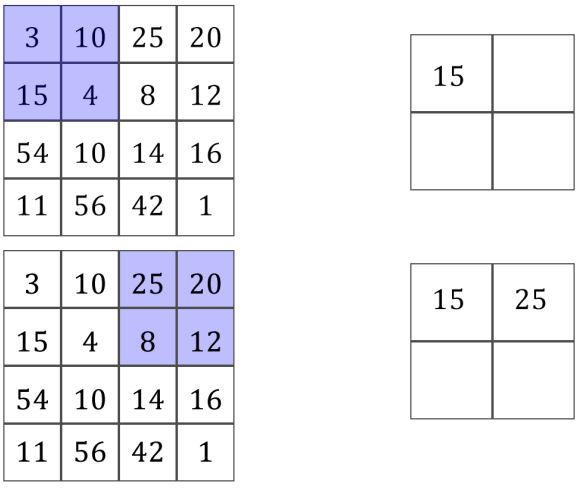

Figura 11: Representación de la operación de pooling

Se crea un archivo en donde se define la función de pooling, la función recibe como parametros una matriz, un entero que

representa el tamaño de la ventana del pooling y un entero que indica el desplazamiento de la ventana.

```
import numpy as np

def max_pooling(matrix, size, stride):
    h,w = matrix.shape #input matrix shape
    if h%2!=0: #If odd
        h=h+1
    if w%2!=0:
        w=w+1
    pooled_matrix = np.zeros((int(h/2),int(w/2))) #Final matrix size
    movey = 0
    for i in range(pooled_matrix.shape[0]):
        movex = 0
        for j in range(pooled_matrix.shape[1]):
            pooled_matrix[i,j] = (matrix[movey:movey+size,movex:movex+size]).max() #Check for maximum
    value
            movex += stride
            movey += stride #steps
    return pooled_matrix
```

4.3. Capas ReLU

Las capas de ReLU tienen como objetivo eliminar los números negativos y convertirlos en cero, de esta manera se dejan las caracteristias importantes de la matriz o imagen. Se crea un archivo y se define la función, de manera que para cualquier elemento de una matriz que sea menor a cero este se convierte en cero.

```
def RELU(matrix):
    new_matrix = matrix.copy()
    new_matrix[new_matrix < 0] = 0
    return new_matrix
```

4.4. Red multicapa

La parte final de una red convolucional es una red neuronal multicapa totalmente conectada. Se crea un archivo que contiene la clase "NeuralNetwork". La clase contiene una serie de metodos que implementa los algoritmos necesaarios para su funcionamiento, estos metodos son:

Método `init`: Inicializa la red neuronal, se calcula el número de capas de la red, y se asignan los demás parametros a variables de instancia. Contiene dos parametros opcionales que son los pesos y los sesgos iniciales de la red, en caso de que la clase no sea instanciada con estos metodos se ejecutan los metodos `weightsInit` y `biasesInit`.

Método `weightsInit` y `biasesInit`: Genera un serie de números aleatorios con distribución normal que son guardados las matrices de pesos y sesgos, estas matrices son guardadas despues en una lista.

Método `train`:Este método contiene una serie de llamadas a otros metodos para realizar el entrenamiento, especificamente llama los metodos `forwardPass`, `backwardPass`.

Método `forwardPass`: Comprueba que función de activación fue asignada a cada capa y se realiza las operaciones correspondientes para generar un salida de la red. La salida de cada capa es guardada en una lista.

Método `backwardPass`: Este es el método más complejo de la red, en este se realiza el algoritmo de backpropagation. Primero calcula el error de la red, inicia una iteración para todas las capas y genera una matriz jacobiana que contiene los resultados de aplicar la función derivada de cada capa a los ultimos resultados de la red. Cada matriz jacobiana es guardada en una lista. Dentro de la iteración se comprueba si se encuentra en la primera iteración, de ser así se calcula las sensibilidaes de los pesos de la ultima capa la cual esta directamente relacionada con el error. Estas sensibilidades son guardadas en una lista. En caso de no ser la primera iteración se utiliza la matriz jacobiana y las sensibilidades de las capas anteriores para calcular como se propaga el error. Las ecuaciones correspondientes para el calculo de las sensibilides son las siguientes: Para el calculo de la salida de la red:

$$a^0 = p \tag{4}$$

Donde a es el vector de resultados y p es el vector de entradas.

$$a^{m+1} = f^{m+1}(W^{m+1}a^m + b^{m+1}) \quad para \quad m = 0, 1, ..., M-1 \tag{5}$$

Donde a^{m+1} es el vector de resultado de la siguiente capa, f^{m+1} es la función de activación de la siguiente capa, W^{m+1} es la matriz de pesos que conecta con la siguiente capa, a^m es el vector de resultados de la capa anterior, b^{m+1} es el vector de sesgos de la siguiente capa, m corresponde a la capa acutal, y M corresponde al número de capas totales. De esta manera el vector de resultados final es:

$$a = a^M \tag{6}$$

Para el calculo de las sensibilidades se empieza por la capa final y termina en la capa inicial. De esta manera la sensibilidades de la capa final se calculan con:

$$s^M = -2F'^M(n^M)(t-a) \tag{7}$$

Donde s^M es la matriz de sensibilidades de la ultima capa t es el vector de resultado esperado, a es el vector actual de resultado y F'^M es la jacobiana definida a partir de las funciones de activación de la ultima capa que se expresa como:

$$F'^m(n^m) = \begin{bmatrix} f'^m(n_1^m) & 0 & \dots & 0 \\ 0 & f'^m(n_2^m) & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & f'^m(n_{S^m}^m) \end{bmatrix}, \tag{8}$$

Para el calculo de las demás sensibilidades se usa:

$$s^m = F'^m(n^m)(W^{m+1})^T s^{m+1}, \quad para \quad m = M-1, ..., 2, 1. \tag{9}$$

Método `weightsUpdate`: En este método se realiza la actualización de los pesos, utilizando las sensibilides calculadas. Para esto se usa:

$$W^m(k+1) = W^m(k) - \alpha s^m (a^{m-1})^T \tag{10}$$

En donde $W^m(k+1)$ es la nueva matriz de pesos, $W^m(k)$ es la vieja matriz de pesos, α es la tasa de aprendizaje, s^m es la matriz de sensibilidaes de la capa m, $(a^{m-1})^T$ es el vector de resultados de la capa transpuesto.

Método `predict`: Este método se utiliza para utilizar la red neuronal para procesar una entrada y obetener un respuesta.

Los metodos `sigmoid`, `linear`, `RELU`, `softmax`, `tanh` se refieren a los métodos es los que se definen las distintas funciones de activación que pueden ser utilizadas en la red.

Método `jacobian`: Este método genera la matriz jacobiana tomando en cuenta la función de activación de la capa en la que se encuentra.

El código correspondiente a la implementación desde cero de la red neuronal multicapa utilizando solamente numpy para el manejo de matrices es el siguiente:

```
import numpy as np

class NeuralNetwork():
    def __init__(self,neurons,activation,error_function,learning_rate, weights=None,biases=None):
        self.layers = len(neurons) - 1 #number of layers
        self.neurons = neurons #number of neurons per layer
        self.activation = activation
        self.error_function = error_function #Error function
        self.weights = weights
        self.biases = biases
        self.learning_rate = learning_rate
        self.jacobians_list = []
        if not self.weights:
            self.weightInit()
        if not self.biases:
            self.biasesInit()

    def weightInit(self):
        self.weights = []
        for i in range(self.layers):
            self.weights.append(np.random.normal(0, 0.01,size=(self.neurons[i+1],self.neurons[i])))

```

```
23      def biasesInit(self):
24          self.biases = []
25          for i in range(self.layers):
26                  self.biases.append(np.random.normal(0,0.01,size=(self.neurons[i+1],1)))
27
28      def train(self,inputs, target):
29          self.outputs = [inputs]
30          self.deltas = []
31          self.forwardPass()
32          self.backwardPass(target)
33          self.weightsUpdate()
34
35      def forwardPass(self):
36          for i in range(self.layers):
37              if self.activation[i] == "sigmoid":
38                  self.outputs.append(self.sigmoid(self.weights[i]*self.outputs[-1] + self.biases[i]))
39              elif self.activation[i] == "linear":
40                  self.outputs.append(self.linear(self.weights[i]*self.outputs[-1] + self.biases[i]))
41              elif self.activation[i] == "RELU":
42                  self.outputs.append(self.RELU(self.weights[i]*self.outputs[-1] + self.biases[i]))
43              elif self.activation[i] == "softmax":
44                  self.outputs.append(self.softmax(self.weights[i]*self.outputs[-1] + self.biases[i]))
45              elif self.activation[i] == "tanh":
46                  self.outputs.append(self.tanh(self.weights[i]*self.outputs[-1] + self.biases[i]))
47
48      def backwardPass(self,targets):
49          self.error = targets - self.outputs[-1]
50          for i in range(self.layers):
51              jacobian = self.jacobian(self.neurons[-1-i], self.activation[-1-i],self.outputs[-1-i])
52              self.jacobians_list.append(jacobian)
53              if i == 0:
54                  if self.error_function == "SE":
55                      s = -2 * jacobian * (targets - self.outputs[-1-i])
56                  elif self.error_function == "MSE":
57                      s = jacobian * (self.outputs[-1-i] - targets)
58              else:
59                  rhs = self.weights[-i].T * self.deltas[-1]
60                  s = self.jacobians_list[-1] * rhs
61              self.deltas.append(s)
62
63      def weightsUpdate(self):
64          for i in range(self.layers):
65              self.weights[self.layers-1-i] -= self.learning_rate * self.deltas[i] * self.outputs[-2-i].T
66              self.biases[self.layers-1-i] -= self.learning_rate * self.deltas[i]
67
68      def predict(self,input):
69          for i in range(self.layers):
70              if self.activation[i] == "sigmoid":
71                  input = self.sigmoid(self.weights[i]*input + self.biases[i])
72              elif self.activation[i] == "linear":
73                  input = self.linear(self.weights[i]*input + self.biases[i])
74              elif self.activation[i] == "RELU":
75                  input = self.RELU(self.weights[i]*input + self.biases[i])
76              elif self.activation[i] == "softmax":
77                  input = self.softmax(self.weights[i]*input + self.biases[i])
78              elif self.activation[i] == "tanh":
79                  input = self.tanh(self.weights[i]*input + self.biases[i])
80          return input
81
82      def sigmoid(self,x,deriv = False):
83          if deriv:
84              return x * (1 - x)
85          return 1/(1 + np.exp(-x))
86
87      def linear(self,x, deriv = False):
88          if deriv:
89              return 1
90          return x
91
92      def RELU(self,x, deriv=False):
93          if deriv:
```

```
            return np.where(x < 0, 0, 1)
        return np.where(x < 0, 0, x)

    def softmax(self,x, deriv = False):
        if deriv:
            return 1 / (1 + np.exp(-x))
        return np.log(1 + np.exp(x))

    def tanh(self,x,deriv=False):
        if deriv:
            return 1 - (x * x)
        return np.tanh(x)

    def jacobian(self,neurons,activation,x):
        result = np.zeros((neurons,neurons))
        x_counter = 0
        for i in range(neurons):
            for j in range(neurons):
                if i == j:
                    if activation == "sigmoid":
                        result[i,j] = self.sigmoid(x[x_counter],deriv=True)
                    elif activation == "linear":
                        result[i,j] = self.linear(x[x_counter],deriv=True)
                    elif activation == "RELU":
                        result[i,j] = self.RELU(x[x_counter],deriv=True)
                    elif activation == "softmax":
                        result[i,j] = self.softmax(x[x_counter],deriv=True)
                    elif activation == "tanh":
                        result[i,j] = self.tanh(x[x_counter],deriv=True)
                    x_counter += 1
        return result
```

La clas de la red neuronal recibe como parametros de instanciación, una lista que contiene el número de neuronas por capa, una lista que contiene las funciones de activación de las capas, un string que indica la función de error a utilizar y la tasa de aprendizaje. Los parametros para instanciar la red con pesos y sesgos iniciales son opcionales, y como se menciono anteriormente esos se inicializan automaticamente con los métodos `weightsInit` y `biasesInit`. Un ejemplo de instanciación de la clase de la red neuronal es el siguiente:

```
neurons = [2,4,4,2] #Number of neurons
activation = ["RELU","RELU","RELU","sigmoid"] #Activation functions
error = "SE" #Squared error function
lr = 0.1 #Learning rate
NN1 = NeuralNetwork(layers,neurons, activation,error,weights,biases, lr)
```

La red esta adaptada para manejar cualquier número de capas y neuronas. Este ejemplo de instanciación corresponde a la siguiente arquitectura mostrada en la figura 12:

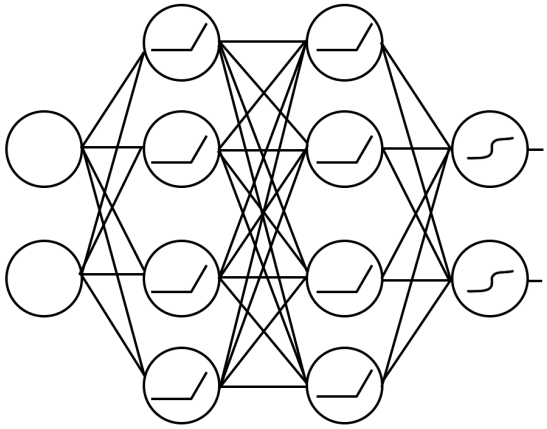

Figura 12: Red neuronal del ejemplo de la instancia

Por ultimo se tiene un ultimo archivo que contiene la clase para las capas convolucionales, esta clase solo hace uso de la función de convolución definida anteriormente y hace el proceso de agrupar los resultados de aplicar todos los kernel:

```
class ConvNet():
def __init__(self,layers):
    self.layers = layers

def process(self,inputImage):
    original_image = inputImage.copy()
    neurons_results = []
    for j in self.layers:
        inputImage = original_image.copy()
        for i in j:
            if i[0] == "Convolution":
                inputImage = Convolutions.crossCorr2d(inputImage, i[1], padding=i[2])
            elif i[0] == "Pooling":
                inputImage = Pooling.max_pooling(inputImage,i[1],i[2])
            elif i[0] == "RELU":
                inputImage = RELU.RELU(inputImage)
        neurons_results.append(inputImage)
    return np.array(neurons_results)
```

4.5. Arquitectura de la red neuronal del problema

El resultado de experimentar con distintas arquitecturas es una red neuronal con una capa de convolución sin padding, con los kernerls diagonales, verticales y horizontales. El resultado de la capa de convolución es un vector de tamaño 1024×1 el cual es usado como entrada a la capa densa. Las capa densa esta formada por la capa de entradas que se dirige hacia la capa de salida, que da como resultado un vector de tamaño 26×1, las posiciones de este vector representan cada una de las letras del abecedario en ingles. Esta ultima capa utiliza una función de activación sigmoidal, lo que regresa la probabilidad de que la entrada sea una de las 26 letras. Se utiliza una tasa de aprendizaje de 0.5 y se entrena por 500,000 iteraciones. Los pesos y los sesgos son guardados en archivos de texto al instanciar la red y utilizados actualizados despues de ser entrenada. El código correspondinete a la inicialización de la red neuronal es le siguiente:

```
import numpy as np
import getData
import NeuralNetworks as NN
import sys
#Architecture
n = 1024
neurons = [n,26]
activation = ["sigmoid"]
error = "SE"
learning_rate = 0.3
NN1 = NN.NeuralNetwork(neurons, activation, error, learning_rate)
getData.storeData("weights.txt", NN1.weights,"w")
getData.storeData("biases.txt", NN1.biases,"w")
file = open("architecture.txt","w")
for i in range(len(neurons)):
    string = str(neurons[i])+","
    if i == len(neurons) - 1:
        string = str(neurons[i])
    file.write(string)
file.write("\n")
for i in range(len(activation)):
    string = activation[i]+","
    if i == len(activation) - 1:
        string = activation[i]
    file.write(string)
file.write("\n")
file.write(error+"\n")
file.write(str(learning_rate)+"\n")
file.close()
```

En el código se puede observar también los bloques correspondientes a la escritura de los pesos en archivos. El archivo correspondiente al entrenamiento de la red neuronal es el siguiente:

```
import numpy as np
import sys
import getData
import NeuralNetworks as NN
import ConvNet as CN
import Kernels

weights = getData.readData("weights.txt")
biases = getData.readData("biases.txt")
neurons, activation, error, learning_rate = getData.readArchitecture("architecture.txt")
NN1 = NN.NeuralNetwork(neurons, activation, error, learning_rate, weights=weights, biases=biases)
#Preparing input data
inputs = getData.readData("trainingData.txt")
targets = getData.readData("targetData.txt")
n = 18
newInputs = []
for i in range(len(inputs)):
    counter = 0
    newMatrix = []
    for j in range(n):
        newRows = []
        for k in range(n):
            newRows.append(inputs[i][counter].item())
            counter += 1
        newMatrix.append(newRows)
    newMatrix = np.asarray(newMatrix)
    newInputs.append(newMatrix)

# Intantiate Conv Layers
padding = False
ConvLayers = CN.ConvNet(
    [
        [["Convolution", Kernels.diagonal_left, padding] ],
        [["Convolution", Kernels.diagonal_right, padding]],
        [["Convolution", Kernels.vertical_line, padding]],
        [["Convolution", Kernels.horizontal_line, padding]]
    ]
    )

iterations = 500000
print(len(inputs))
for i in range(iterations):
    actual_input = ConvLayers.process(newInputs[i %len(inputs)])
    actual_input = actual_input.flatten()
    actual_input = actual_input.reshape((len(actual_input),1))
    actual_target = targets[i %len(targets)]
    NN1.train(actual_input, actual_target)
    print("MSE:", np.power(NN1.error,2).sum()/len(NN1.error))
    print(i)
print("Trained")
getData.storeData("weights.txt", NN1.weights,"w")
getData.storeData("biases.txt", NN1.biases,"w")
```

Como se puede observar, este archivo implementa cada uno de los archivos anteriormente mencionados, el proceso de entrenamieneto se lleva acabo en un ciclo final que dura la cantidad de iteraciones asignada. Al terminar el entrenamiento los pesos y los sesgos se escriben directamente en los archivos Una vez teniendo esto solo es cuestión de que la aplicación web envié una petición al servidor y ejecute el siguiente script de python:

```
import numpy as np
import sys
import getData
import NeuralNetworks as NN
import ConvNet as CN
import Kernels
import string as st
try:
    #Preparing input data
    input_data = getData.parseRequestData(sys.argv[1], sys.argv[2], "c")
    # Intantiate Conv Layers
```

```
    padding = False
    ConvLayers = CN.ConvNet(
        [["Convolution", Kernels.diagonal_left, padding] ],[["Convolution", Kernels.diagonal_right, padding]],
        [["Convolution", Kernels.vertical_line, padding]],[["Convolution", Kernels.horizontal_line, padding]]])
    result = ConvLayers.process(input_data)
    result = result.flatten()
    result = result.reshape((len(result),1))
    #Intantiate Network from saved files
    weights = getData.readData("./backend/weights.txt")
    biases = getData.readData("./backend/biases.txt")
    neurons, activation, error, learning_rate = getData.readArchitecture("./backend/architecture.txt")
    NN1 = NN.NeuralNetwork(neurons, activation, error, learning_rate, weights=weights, biases=biases)
    print(st.ascii_lowercase[np.argmax(NN1.predict(result))])
except Exception as e:
    print("Final error:",e)
```

Una vez ejecutado el proceso de python el servidor recibe la respuesta de la red, y la envía al fronend de la aplicación donde muestra la letra que la red predijo.

4.6. Conclusiones

El proyecto se trabajó en parejas, por un lado la compañera Sonia Alejandra Treviño Rivera se encargó de la aplicación web, del preprocesamiento de datos entre el servidor y python y de la generación del conjunto de datos de entrnamiento de la red. Mientras que el compañero Edson Raul Cepeda Marquez se encargo de desarrollar el código correspondiente a la red neuronal y los algoritmos correspondientes. Los problemas principales eran muy variados, desde fallás en la lógica de la programación de los algoritmos, hasta problemas de comunicación entre la aplicación y el servidor. Estos problemas fueron resueltos en equipo recibiendo retroalimentación de cada uno e ideas para la resolución de estos problemas. Una de las cosas que se puede compartir de esto, es que desarrollar los algoritmos de aprendizage automatico desde cero no es lo lo más eficiente, pero en cuestiones didacticas deja unas buenas bases del tema, lo que despues se traduce en la facilidad para usar librerias. Todo lo expuesto a lo largo del documento fue comprendido en su totalidad, por lo que los conocimientos adquiridos se ven reflejados en este documento. Debido a la falta de tiempo, el conjunto de datos no es lo que esperabamos, se planeaba por lo menos 100 ejemplos por cada letra para que la red neuronal fuera entrenada de manera correcta, pero conseguimos hacer 30 por cada letra. Lo que de todos modos resulto en un porcentaje de exito del 80 % de las predicciones de la red.

Referencias

[1] Hagan, M. (2014). Neural Network Design. Ok, Estados Unidos.

[2] Poole, D. (2015) Linear Algebra: A modern introduction. Stamford, CT.

[3] Pokharna, H. (2016, 28 de julio). The best explanation of Convolutional Neural Networks. Recuperado de: `https://medium.com/technologymadeeasy/the-best-explanation-of-convolutional-neural-networks-on-the-internet-fbb8b1ad5df8`

[4] Sharma, S. (2017, 6 de septiembre). Activation Functions in Neural Networks Recuperado de: `https://setosa.io/ev/image-kernels/`

[5] Powell, V. (2018). Image Kernels explained Visually Recuperado de: `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`

[6] Edson Cepeda, Neural Networks, Github repository: `https://github.com/OrbitalCardinal/NeuralNetworks`