Universidad Veracruzana

Maestría en Inteligencia Artificial
Visión por computadora

Tarea 2
Escalado de imágenes mediante vecino más cercano e interpolación bilineal

Ulises Jiménez Guerrero

27 de febrero de 2025

Reporte tarea 2

1. Objetivos

- Crear un programa de computadora capaz de escalar imágenes, ya sea aumentando o encogiendo su tamaño, utilizando las técnicas de vecino más cercano e interpolación bilineal. La entrada del programa será las dimensiones de la imagen deseada, y su salida las imágenes resultantes con estos métodos.
- Comprobar el funcionamiento del programa y de cada método utilizando la imagen de prueba proporcionada para ello.

2. Metodología

2.1. Materiales utilizados

Para el desarrollo se utilizó el lenguaje de programación Python, versión 3.12.8. Para el manejo de las imágenes se utilizó la paquetería de código libre opencv para Python, junto a la paquetería Numpy para el manejo de las matrices generadas por las imágenes. Finalmente, se utilizó la paquetería de Matplotlib para el despliegue de imágenes en la pantalla, que a su vez instala varias dependencias. Todas las paqueterías junto a las versiones utilizadas se encuentran listadas en el anexo. Para la comprobación del algoritmo, se utilizó la imagen 2.19 (a) del libro [1] , mostrada en la imagen 1. Esta imagen tiene un tamaño original de 1024×1024 píxeles, y se utilizó en una escala de grises con 8 bits.

Figura 1: Imagen utilizada para probar los algoritmos. Muestra una rosa blanca sobre un fondo negro, con detalles del tallo. No se encuentra a escala.

2.2. Algoritmos e implementación

2.2.1. Método del vecino más cercano

Este es el método más sencillo para escalar el tamaño de una imagen digital, y por esto da los peores resultados. Su funcionamiento es sencillo: para cada nuevo píxel de la imagen escalada, se le asigna el valor de la tonalidad del píxel más cercano en la imagen original. Para ello, se debe realizar un mapeo de las coordenadas de la imagen reajustada a la imagen original. Este problema se solventa utilizando la proporción entre las dimensiones de la imagen original y la imagen deseada. De esta forma, un píxel con coordenadas (i, j) en la imagen ajustada tiene las coordenadas (x, y) en la imagen original, dadas por:

$$x = i \times r_w, \quad y = j \times r_h \quad \text{donde} \quad r_w = \frac{\text{Ancho original}}{\text{Nuevo ancho}}, \quad r_h = \frac{\text{Alto original}}{\text{Nuevo alto}}.$$

Como el cálculo de distancias es un proceso costoso, se realiza una técnica diferente para obtener el píxel más cercano. Se redondea el valor de las coordenadas (x, y) obtenidas y se utiliza la intensidad del píxel $f(x, y)$ con las coordenadas redondeadas. Debido a que existen varios métodos para el redondeo de números, esto puede ocasionar resultados diferentes, pero no a un nivel significativo [2].

Para la implementación realizada se utilizó la función *int()* de Python, que realiza un redondeo hacia abajo. Este es el mismo redondeo utilizado en la función de ajuste de imágenes nativa de opencv con el método de vecino más cercano. El algoritmo se aplicó de forma directa, generando una matriz vacía con el tamaño indicado por el usuario y aplicando la fórmula sobre cada píxel, recorriendo la imagen de izquierda a derecha y de arriba a abajo. Cabe destacar que este tipo de ciclos anidados son lentos en Python, donde se recomienda realizar operaciones matriciales en Numpy para mejorar la velocidad de los programas.

2.2.2. Método de interpolación bilineal

Este método es más refinado que el de vecino más cercano, y presenta resultados más suaves y que preservan una mayor calidad de imagen. Sin embargo, esto lo hace más costoso, requiriendo un tiempo de procesado mayor.

Se empieza con un mapeo de las coordenadas de la nueva imagen a la imagen original, de la misma forma que en el algoritmo anterior. Dada la posición en la imagen original, se obtiene la tonalidad realizando un promedio ponderado en base a los cuatro píxeles más cercanos. Estos se encuentran en la esquina superior derecha, esquina superior izquierda, esquina inferior derecha y esquina inferior izquierda. Para obtener el promedio ponderado se realiza una interpolación tanto en la dimensión X como en la Y, esto es, sobre dos líneas, dando su nombre al algoritmo.

Dado que cada píxel tiene un valor de una unidad de medida, se puede considerar que se trabaja en un cuadrado unitario. Tomando este sistema de referencia, la esquina superior izquierda tiene coordenada $(1, 0)$, la inferior derecha $(0, 1)$, etc. De esta forma, se tiene la siguiente ecuación para la intensidad del píxel en las coordenadas (x, y), realizando ambas interpolaciones en un solo cálculo:

$$f(x,y) = (f(1,0) - f(0,0))\,x + (f(0,1) - f(0,0))\,y + (f(1,1) - f(0,0) - f(0,1) - f(1,0))\,xy + f(0,0) \quad (1)$$

La implementación se realizó de forma directa, generando una matriz vacía con las dimensiones adecuadas y recorriendo cada píxel, calculando su intensidad mediante el método indicado. Se realizaron algunas modificaciones a 1, sugeridas en [3] para simplificar cálculos. Sin embargo, sigue siendo una operación costosa, tomando varios segundos para procesar imágenes de tamaños mayores a 1000×1000 píxeles. Para la obtención de los píxeles vecinos más cercanos, se utilizó la función *floor()* de numpy para obtener un redondeo hacia abajo, dando los valores inferiores en los ejes X y Y. Para los valores superiores, se sumo 1 a estos resultados. Sin embargo, se debe prestar especial atención a los píxeles ubicados en los bordes de la imagen. Existen diversas técnicas para lidiar con ellos, como realizar una copia del borde alrededor de la imagen. En nuestro caso, se utilizó la función *min()* para asegurarnos que la coordenada obtenida siempre se mantiene dentro de los límites de la imagen, aunque esto resulte en que se repita el mismo píxel en algunos cálculos.

2.2.3. Manejo de las imágenes

Con el uso de la función *imread()* de opencv, se leyeron a las imágenes como matrices de numpy, fáciles de manipular en Python. Se estableció el canal de color como escala de grises, de manera que se utilizaron matrices de tamaño (Pixeles de ancho $\times$ Pixeles de alto), donde el valor de cada celda se corresponde a la intensidad de la tonalidad de gris en ese píxel. Las matrices resultantes de los algoritmos se guardaron como imágenes formato PNG, lo que facilita su inclusión en el reporte. Además, con el uso de matplotlib se implementó la opción de visualizar las imágenes resultantes junto a la imagen original, ajustando el tamaño del píxel para que ocupen el mismo espacio en la pantalla, lo que permitió una mejor comparación de la calidad de imagen resultante.

3. Resultados

Se inició contrayendo la imagen en 1 a un tamaño de 256×256 píxeles, esto es, un cuarto de su tamaño. Las imágenes resultantes se pueden observar en 2. La pérdida de calidad en la imagen es evidente para ambos métodos, sobre todo en los bordes de los pétalos de la rosa. No se observa una diferencia en los resultados de los dos métodos, debido a que la pérdida de información es significativa en ambos casos.

(a) Compresión de imagen mediante vecino más cercano. Escala real,

(b) Compresión de la imagen mediante interpolación bilineal. Escala real.

Figura 2: Resultados de la compresión de la imagen de prueba.

Posteriormente, se tomó cada una de las dos imágenes resultantes y se aplicaron ambos métodos para re-escalarlas a su tamaño original de 1024×1024 píxeles. Los resultados se muestran en las figuras 3 y 4. Ninguna de las cuatro imágenes presenta una calidad igual a la original. Esto es de esperarse, dado que el proceso de interpolación siempre pierde información y agrega ruido a la imagen, efecto que se va sumando mientras más veces se realice este proceso sobre la misma imagen. De todas las imágenes obtenidas, la que tiene un resultado más suave es aquella que se sometió al proceso de interpolación bilineal dos veces, una para compresión y otra para expansión. Aunque en una revisión rápida pueda parecer que su calidad no difiere significativamente de 1, una revisión más detallada muestra la presencia de ruido en los bordes del pétalo, además de una pérdida de la suavidad de su forma y un cambio más brusco en la tonalidad. Métodos más sofisticados como interpolación bicúbica permiten realizar este proceso de forma más eficaz, pero la pérdida de información inherente a los métodos de interpolación no se logra eliminar.

4. Conclusiones

El método de interpolación mediante vecino más cercano es sencillo de implementar y de calcular, lo que lo hace ideal para situaciones donde se busca un ajuste mínimo de las dimensiones de una imagen. Para estos casos, la pérdida de información es despreciable, y su uso es adecuado. En cambio, cuando el cambio en el tamaño de la imagen original es significativo, se requieren de métodos más sofisticados como la interpolación bilineal para mantener la calidad de imagen original lo más posible. Sin embargo, para cambios extremos como el realizado en este trabajo, este método sigue presentando defectos notables a simple vista, lo que indica que se requieren técnicas todavía más avanzadas como la interpolación bicúbica. Un trabajo a futuro sería el comparar el desempeño de este algoritmo con los ya implementados.

Algo que destacó este trabajo es que se debe evitar redimensionar la misma imagen múltiples veces, dado que el efecto de la pérdida de información se suma en cada iteración, y la calidad de la imagen decae rápidamente. En casos donde se requiera la misma imagen en diferentes resoluciones, siempre se debe aplicar la transformación sobre la imagen original, minimizando de esta manera la pérdida de calidad.

Referencias

[1] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Pearson, New York, fourth edition, global edition edition, 2017.

[2] Krystian Wojcicki. Nearest Neighbour Interpolation. https://kwojcicki.github.io/blog/NEAREST-NEIGHBOUR, August 2020.

[3] Krystian Wojcicki. Bilinear interpolation. https://kwojcicki.github.io/blog/BILINEAR-IMAGE-INTERPOLATION, August 2022.

(a) Expansión de la imagen mediante vecino más cercano.

(b) Expansión de la imagen mediante interpolación bilineal.

Figura 3: Resultados de la expansión de la imagen comprimida obtenida mediante el método del vecino más cercano.

(a) Expansión de la imagen mediante vecino más cercano.

(b) Expansión de la imagen mediante interpolación bilineal.

Figura 4: Resultados de la expansión de la imagen comprimida obtenida mediante el método de interpolación bilineal.

5. Anexo

5.1. Lista de paqueterías y versiones

- contourpy==1.3.1
- cycler==0.12.1
- fonttools==4.56.0
- kiwisolver==1.4.8
- matplotlib==3.10.0
- numpy==2.2.3
- opencv-python==4.11.0.86
- packaging==24.2
- pillow==11.1.0
- pyparsing==3.2.1
- python-dateutil==2.9.0.post0
- six==1.17.0

5.2. Código

```
import numpy as np
import matplotlib.pyplot as plt
import cv2 as cv

def image_resize(img_path:str, new_width:int, new_height:int,
                 show_result:bool = False):
    '''Function used to change the size of an image by the nearest
    neighbor and bilinear interpolation methods.

    It can show the original images and the two resulting ones.

    img_path: location path to the image \\
    new_height: desired height for the resulting image \\
    new_width: desired width for the resulting image \\
    show_result: show images for comparison. Default is False

    returns: (img using nearest neighbor, img using bilinear interpolation
       )'''

    # The modifier 0 indicates that the image must be read in grayscale
    img = cv.imread(img_path, 0)
```

```
    # We obtain the width and height from the original image
    height, width = img.shape

    # A new empty matrix is created with the desired dimensions. It's
       gonna be
    # filled with the corresponding values based on the algorithm being
       used.
    # The data type is uint8, so the values are in the range [0,255]
    new_img_nn = np.empty((new_height, new_width), dtype=np.uint8)
    new_img_bi = np.empty((new_height, new_width), dtype=np.uint8)

    # We calculate the ratio within the original dimensions of the image
       and
    # the desired ones
    height_ratio = height/new_height
    width_ratio = width/new_width

    # NEAREST NEIGHBOR METHOD.
    for row in range(new_height):
        for column in range(new_width):
            # Going from the new coordinates to the original ones using
               the
            # corresponding ratio. The int() function rounds the number
               down,
            # giving us the nearest pixel
            x = int(column * width_ratio)
            y = int(row * height_ratio)
            # Using the obtained values in the resized image
            new_img_nn[row, column] = img[y,x]

    # BILINEAR INTERPOLATION METHOD.
    for row in range(new_height):
        for column in range(new_width):
            # i = row, j = column

            # Going from the new coordinates to the original ones.
            x = column * width_ratio
            y = row * height_ratio

            # Calculating the position of the four nearest pixels.
            x_low, y_low = int(np.floor(x)), int(np.floor(y))
            # We need to consider the edge cases using the min() function,
                to
            # stay inside the bounds of the image
            x_top, y_top = min(x_low + 1, width - 1), min(y_low + 1,
               height - 1)

            # Differences required in the equation
            x_diff = x - x_low
            y_diff = y - y_low

            # Coordinates of the closest pixels
            top_left = img[y_low, x_low]
            top_right = img[y_low, x_top]
```

```
            bottom_left = img[y_top, x_low]
            bottom_right = img[y_top, x_top]

            # Bilinear interpolation formula
            inter_value = ((1 - x_diff) * (1 - y_diff) * top_left +
                           x_diff * (1 - y_diff) * top_right +
                           (1 - x_diff) * y_diff * bottom_left +
                           x_diff * y_diff * bottom_right)

            new_img_bi[row, column] = inter_value.astype(np.uint8)

    # Shows the original images and the resized ones for comparison,
       adjusting
    # the pixel size so they have an equal size on the screen
    if show_result:
        fig, (ax1,ax2,ax3) = plt.subplots(1, 3)
        ax1.imshow(img, cmap='gray')
        ax1.set_title('Original image')
        ax1.axis('off')
        ax2.imshow(new_img_nn, cmap='gray')
        ax2.set_title('Nearest neighbor method')
        ax2.axis('off')
        ax3.imshow(new_img_bi, cmap='gray')
        ax3.set_title('Bilinear interpolation method')
        ax3.axis('off')
        plt.show()

    return new_img_nn, new_img_bi


if __name__=='__main__':
    result = image_resize('shrinked_img_nn.png', 1024, 1024)
    # cv.imwrite('expanded_img_bi2nn.png', result[0])
    # cv.imwrite('expanded_img_bi2bi.png', result[1])
```