UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA - UTEC

Departamento de Ciencias de la Computación
Prof. Cristhian López del Álamo

Proyecto 4: Autoencoders

https://github.com/juancarloscruzd/utec-ml-proyecto-4

Fernando Alarcón Delgado
fernando.alarcon@utec.edu.pe

Juan Carlos Cruz Diaz
juan.cruz@utec.edu.pe

Victor Hugo Herrera Asmat
victor.herrera@utec.edu.pe

I. INTRODUCCIÓN

En este proyecto, se desarrolla un modelo, basado en autoencoders, capaz de transformar imágenes de baja resolución a imágenes en alta resolución a través de un *dataset* conteniendo una carpeta *Train*, con imágenes de ambos tipos, y una carpeta *Val*, para validation y *Testing*, también, con ambos tipos de imágenes. Se plantea un modelo, y hacen pruebas exhaustivas en cuanto a tamaño de convolución, número de kernels, número de capas (convolution y pooling), además de analizar el comportamiento de la propuesta utilizando tecnicas de normalización.

II. EXPLICACIÓN DE LAS ARQUITECTURAS

II-A. Convolution y Pooling

En el contexto de redes neuronales, ambos conceptos son mecanismos que facilitan y potencian el entrenamiento de dichas redes al retener cierta información de la imagen en diferentes capas, en el caso de las convoluciones, o al reducir los datos a procesar minimizando la pérdida de información, en el caso de *polling*. Estas técnicas han logrado un ámplio desarrollo de las redes neuronales en casos de uso como clasificación de imágenes, detección de objetos, reconocimiento facial entre otros. (LeCun, Kavukcuoglu, y Farabet, 2010)

1. *Convolution*

El concepto de convolución es el aplicar un filtro a una imagen para obtener otra con diferentes características, como se puede observar en la Figura 1.
Teniendo en cuenta que una imagen puede ser representada como una matriz de pixeles, y cada pixel puede representar una intensidad de color. (Figura 2).
El filtro es siempre más pequeño que el *input*, y la operación realizada entre el filtro y el *input* es un producto punto. Usar un filtro más pequeño que la imagen garantiza esta operación de producto punto.
La aplicación sistemática del filtro a través de la imagen hace posible la detección de patrones para los que el filtro está diseñado, por ejemplo, líneas diagonales, horizontales, etc.
El resultado de multiplicar el filtro con la imagen una vez, genera un valor único. Si el filtro es aplicado múltiples veces, el resultado es un array 2d de valores de salida que representan versiones de la imagen, llamado *feature map*.
Una vez creado el feature map, se pueden usar dichos valores en funciones como ReLU, similar a lo que se hace para las salidas de una red neuronal completamente conectada. (Figura 3)

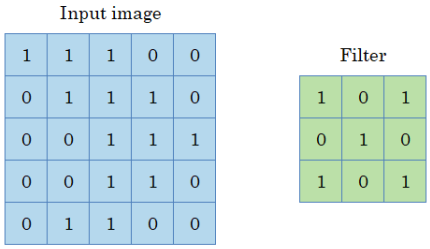

Figura 1. Ejemplo de una imagen vectorizada y un filtro. Fuente: Towards Science

2. *Polling*

Es el proceso por el cual se agrupan pixeles (por ejemplo, grupos de 2x2 pixeles) y aplicar una función de agregación sobre el grupo, la más común es la de tomar el valor máximo del grupo, conocida como ***Max Polling*** (Figura 4). Otra agregación común es la de tomar el promedio, conocida como ***Average Polling***
Como ejemplo, podemos tomar una imagen a tamaño normal (Figura 5). Dicha imagen luego de ser

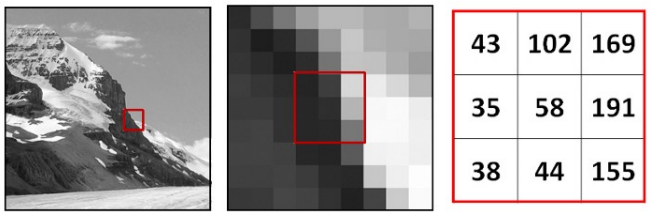

Figura 2. Ejemplo la diferencia de intensidad de color en una imagen. Fuente: Towards Science

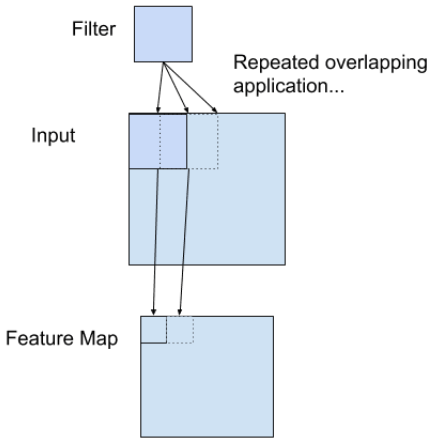

Figura 3. Ejemplo de un filtro aplicada a una imagen para crear un *feature map*. Fuente: Machine Learning Mastery

transformada por un filtro y por un proceso de polling (Figura 6), se observan dos cosas prinicipalmente:

- El tamaño de la imagen, se reduce a la mitad, esto debido a aplicar el max polling en grupos de 2x2 y solo reteniendo el máximo valor
- Los bordes identificados con el filtro de convolución, no solo son mantenidos, también sin intensificados

Lo que significa que se logra reducir la información pero intensificar las características al aplicar los filtros de convolución a la imagen.

II-B. Kernels

Un kernel es una matriz, que se pasa por la imagen y es multiplicada con ella, de tal manera que la salida es una caraterística de la propia imagen. Por ejemplo, considerando dos imágenes de entrada (Figura 7), para la primera imagen, el valor del centro es $3 * 5 + 2 * -1 + 2 * -1 + 2 * -1 + 2 * -1 = 7$. Incrementando de 3 a . Para la segunda imagen, la salida es $1 * 5 + 2 * -1 + 2 * -1 + 2 * -1 + 2 * -1 = -3$. reduciendo su valor de 1 a -3. Claramente, el contraste entre 3 y 1 es incrementado a 7 y -3, que resultan en una imagen más refinada.

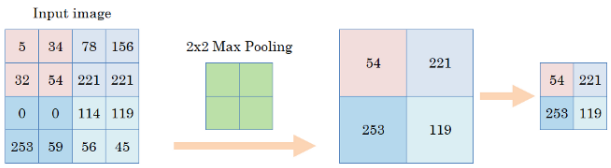

Figura 4. Ejemplo de *max polling*. Fuente: Towards Science

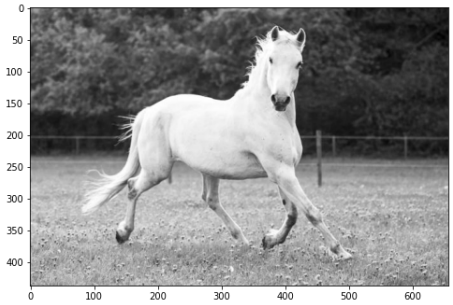

Figura 5. Imagen a original tamaño regular. Fuente: Towards Science

II-C. Funciones de Activación

Las funciones de activación llevan su nombre del proceso similar al que ocurre en el cerebro con las neruronas en donde una señal electrica una vez que alcanza un cierto nivel, dispara la neurona de tal forma que la señal pasa. Si la señal es demasiado baja entonces la neurona no se activa (Wilmott, 2019). Alguna de las funciones de activación más comunes y conocidas son las siguientes:

1. **Linear function:** Sería de la forma:

$$g(x) = x$$

La cual trae el problema al aplicar el *gradient descent* puesto que la derivada siempre es uno así como pierde la naturaleza escencial de la no-linealidad de las redes neuronales.

2. **Step function/Hard limit:** Se comporta de la siguiente manera:

$$g(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

La señal toma una cantidad fija (1) o se atenua (0). Esto es demasiado extremo por que no deja margen a la interpretación probabilistica de la señal. Asi mismo sufre de gradiente ”cero” excepto en un punto donde es infinita.

3. **Positive linear/ReLU function:** Reperesentada por la expresión:

$$g(x) = max(0, x)$$

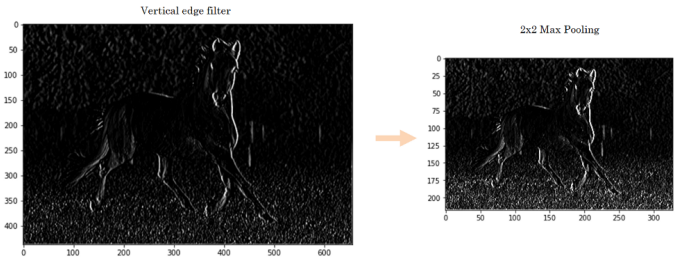

Figura 6. Imagen con filtro y max polling. Fuente: Towards Science

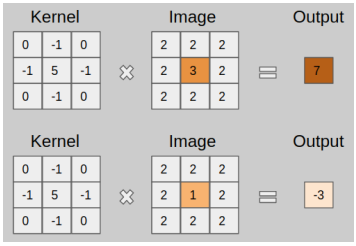

Figura 7. Ejemplo de un kernel aplicado a una imagen. Fuente: Towards Science

ReLU viene del término *Rectified Linear Units*. Es una de las funciones de activación más utilizadas siendo suficientemente no lineal cuando hay varios nodos que interactuan. La señal o pasa sin modificación o se atenua completamente.

4. **Saturating linear function:** Similar a la *Step function* pero no tan extrema.

$$g(x) = \begin{cases} 0 & x < 0 \\ x & 0 \leq x \leq 1. \\ 1 & x > 1 \end{cases}$$

5. **Sigmoid o Logistic functon:** Sige la expresión:

$$g(x) = \frac{1}{1 + e^{-x}}$$

Es una versión gentil de la *Step function*. Es una función de activación muy útil para problemas de clasificación. La funcion *tanh* es también utilizada, pero es simplemente una tranformación lineal de la función logística.

6. **Softmax functon:** Esta función toma un arreglo de K valores $(z_1, z_2, ..., z_K)$ y los mapea en K números entre cero y uno (sumando Uno). Es por tanto una función que convierte varios números en cantidades que tal vez se puedan interpretar como probabilidades.

$$\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

Se utiliza al final, en la capa de salida de la red neuronal, específicamente para problemas de clasificación.

La figura 8 muestra las gráficas de algunas de las funciones de activación más utilizadas en las redes neuronales.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Figura 8. Funciones de activación más usadas. Se muestra su ecuación y su derivada. `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`

III. EXPERIMENTOS

III-A. Modificación del número de Capas (Convolution y Pooling)

A continuación, se presentan los algoritmos *Autoencoders*, tomando en cuenta diferentes arquitecturas. En la Figura 9 se toma en cuenta una arquitectura de dos capas, en la Figura 10 se presenta un arquitectura de 3 capas y; finalmente, en la Figura 11 se presenta un propuesta que toma en cuenta 4 capas.

```
# numero de capas (2 capas)
class Encoder(nn.Module):
  def __init__(self):
    super(Encoder, self).__init__()

    self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=4, stride=2, padding=1) #(256+2*1)= 258-4 =254 -> 254/2+1 = 128
    self.conv2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2, padding=1) #(128+2*1)= 130-4 =126 -> 126/2+1 = 64
    self.fc = nn.Linear(in_features=128*64*64, out_features=256)

  def forward(self, image):
    convs = []
    out = F.relu(self.conv1(image))
    convs.append(out)
    out = F.relu(self.conv2(out))
    convs.append(out)
    out = out.view(out.size(0), -1)
    I = self.fc(out)
    return I, convs

class Decoder(nn.Module):
  def __init__(self):
    super(Decoder, self).__init__()

    self.fc = nn.Linear(in_features=256,out_features=128*64*64)
    self.convTran1 = nn.ConvTranspose2d(in_channels=2*128,out_channels=64, kernel_size=4, stride=2, padding=1) #Modificando inchannels. x2 por la copia U-Net
    self.convTran2 = nn.ConvTranspose2d(in_channels=2*64,out_channels=3, kernel_size=4, stride=2, padding=1) #Modificando inchannels. x2 por la copia U-Net

  def forward(self, latent,params):
    out = self.fc(latent)
    out = out.view(out.size(0), 128, 64, 64)

    out = torch.cat((params[1],out),1)
    out = torch.relu(self.convTran1(out))

    out = torch.cat((params[0],out),1)
    out = torch.tanh(self.convTran2(out))

    return out
```

Figura 9. *Autoencoder* con 2 capas

Respecto a los resultados, a continuación se presentan los hallazgos asociados a cada arquitectura. Como se aprecia en las figuras, la arquitectura que presenta un mayor número de capas permite alcanzar un menor Error Cuadrático Medio.

A continuación, se muestran imagenes de baja resolución y sus respectivas imagenes mejoradas.

```python
# numero de capas (2 capas)
class Encoder(nn.Module):
  def __init__(self):
    super(Encoder, self).__init__()

    self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=4, stride=2, padding=1) #(256+2*1)= 258-4 =254 -> 254/2+1 = 128
    self.conv2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2, padding=1) #(128+2*1)= 130-4 =126 -> 126/2+1 = 64
    self.fc = nn.Linear(in_features=128*64*64, out_features=256)

  def forward(self, image):
    convs = []
    out = F.relu(self.conv1(image))
    convs.append(out)
    out = F.relu(self.conv2(out))
    convs.append(out)
    out = out.view(out.size(0), -1)
    z = self.fc(out)
    return z, convs

class Decoder(nn.Module):
  def __init__(self):
    super(Decoder, self).__init__()

    self.fc = nn.Linear(in_features=256,out_features=128*64*64)
    self.convTran1 = nn.ConvTranspose2d(in_channels=2*128,out_channels=64, kernel_size=4, stride=2, padding=1) #Modificando inchannels. x2 por la copia U-Net
    self.convTran2 = nn.ConvTranspose2d(in_channels=2*64,out_channels=3, kernel_size=4, stride=2, padding=1) #Modificando inchannels. x2 por la copia U-Net

  def forward(self, latent,params):
    out = self.fc(latent)
    out = out.view(out.size(0), 128, 64, 64)

    out = torch.cat((params[1],out),1)
    out = torch.relu(self.convTran1(out))

    out = torch.cat((params[0],out),1)
    out = torch.tanh(self.convTran2(out))

    return out
```

Figura 10. *Autoencoder* con 3 capas

```python
# numero de capas (2 capas)
class Encoder(nn.Module):
  def __init__(self):
    super(Encoder, self).__init__()

    self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=4, stride=2, padding=1) #(256+2*1)= 258-4 =254 -> 254/2+1 = 128
    self.conv2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2, padding=1) #(128+2*1)= 130-4 =126 -> 126/2+1 = 64
    self.fc = nn.Linear(in_features=128*64*64, out_features=256)

  def forward(self, image):
    convs = []
    out = F.relu(self.conv1(image))
    convs.append(out)
    out = F.relu(self.conv2(out))
    convs.append(out)
    out = out.view(out.size(0), -1)
    z = self.fc(out)
    return z, convs

class Decoder(nn.Module):
  def __init__(self):
    super(Decoder, self).__init__()

    self.fc = nn.Linear(in_features=256,out_features=128*64*64)
    self.convTran1 = nn.ConvTranspose2d(in_channels=2*128,out_channels=64, kernel_size=4, stride=2, padding=1) #Modificando inchannels. x2 por la copia U-Net
    self.convTran2 = nn.ConvTranspose2d(in_channels=2*64,out_channels=3, kernel_size=4, stride=2, padding=1) #Modificando inchannels. x2 por la copia U-Net

  def forward(self, latent,params):
    out = self.fc(latent)
    out = out.view(out.size(0), 128, 64, 64)

    out = torch.cat((params[1],out),1)
    out = torch.relu(self.convTran1(out))

    out = torch.cat((params[0],out),1)
    out = torch.tanh(self.convTran2(out))

    return out
```

Figura 11. *Autoencoder* de 4 capas

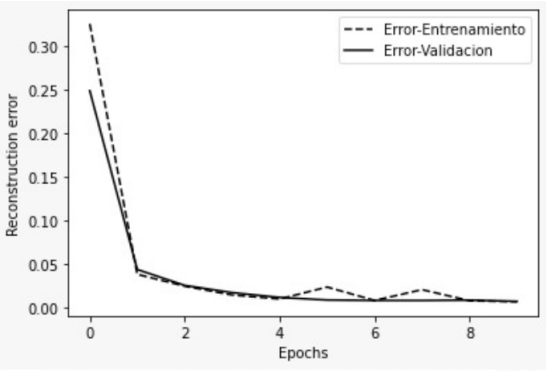

Figura 12. Función de perdidad - *Autoencoder* de 2 capas

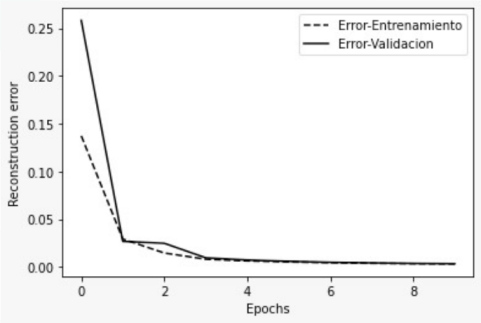

Figura 13. Función de perdidad - *Autoencoder* de 3 capas

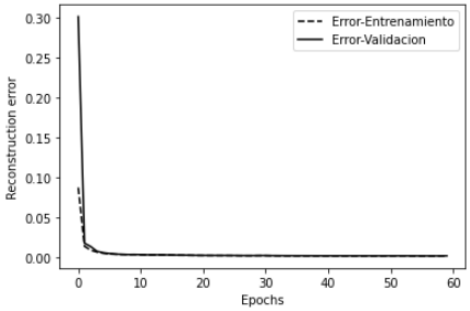

Figura 14. Función de perdidad - *Autoencoder* de 4 capas

III-B. Modificación de tamaños de Kernel

Se realizaron experimentos modificando los tamaños de kernel, para el caso de tamaño 4 y stride 2 (Figura 16) y el caso de tamaño 6 y stride 4 (Figura 17)

III-C. Modificación de la funcion de Activación

III-D. Resultados del Error Cuadrático Medio en diferentes arquitecturas

Cuadro I
RESUMEN DE LOS RESULTADOS.

	2CAP	3CAP	4CAP
ECM	0.007	0.003	0.003

IV. CONCLUSIONES

1. Los resultados muestran que recuperar imagenes de alta calidad es una tarea compleja, por lo cual es recomendable utilizar arquitecturas de varias capas.
2. La modificación de los kernel juega un rol muy importante para identificar detalles en las figuras.
3. Si bien existen numerosas funciones de activación, se recomienda utilizar las funciones tipo relu", debido a

Figura 15. Imagen mejorada - *Autoencoder* de 4 capas

Figura 16. Cambio de Kernel. $kernel_size = 4, stride = 2$

que las mismas permiten que el efecto marginal de la derivada no se desvanezca.

4. Por otro lado, el proceso de estimación de los modelos tomó un tiempo considerable, debido que estos modelos son intensivos en uso de recursos. En este marco, se recomienda utilizar equipos con suficientes recursos para hacer todas las pruebas necesarias.

REFERENCIAS

LeCun, Y., Kavukcuoglu, K., y Farabet, C. (2010). Convolutional networks and applications in vision. En *Proceedings of 2010 ieee international symposium on circuits and systems* (pp. 253–256).

Wilmott, P. (2019). *Machine learning: an applied mathematics introduction* (First edition ed.). Oxford, United Kingdom?: Panda Ohana Publishing.

Figura 17. Cambio de Kernel. $kernel_size = 6, stride = 3$

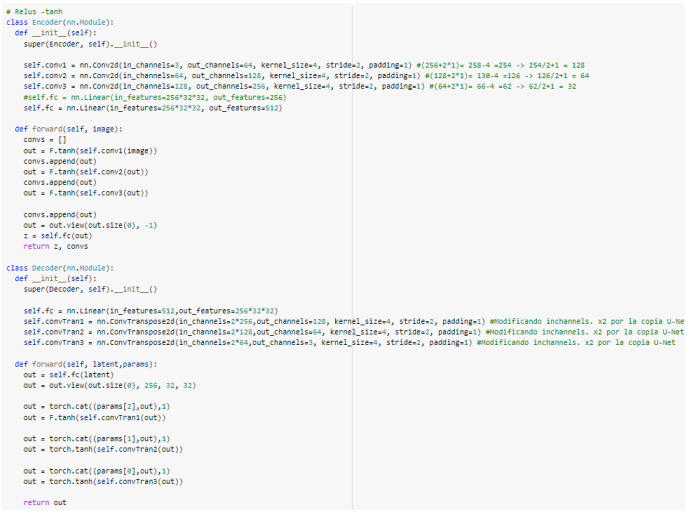

Figura 18. Aplicación de diferentes funciones de activacion