

Clase 4 - Autoenconders y Segmentación

Plan de la presentación

PYTORCH

Autoencoders

Segmentación

PYTORCH

De Wikipedia: " PyTorch¹ es una biblioteca de aprendizaje automático de código abierto basada en la biblioteca de Torch, utilizado para aplicaciones como visión artificial y procesamiento de lenguajes naturales, principalmente desarrollado por el Laboratorio de Investigación de Inteligencia Artificial Facebook (FAIR)."

PyTorch proporciona dos características de alto nivel:

- ▶ Computación de **tensores** (como los array de NumPy) con una aceleración fuerte a través de unidades de procesamientos gráficos (GPU).
- ▶ Permite construir Redes neuronales profundas y utiliza un sistema de diferenciación automática para el cálculo del gradiente (fundamental para el backpropagation).

¹<https://pytorch.org>

¿Qué es un Tensor?

Un **tensor** es la estructura de datos fundamental en PyTorch. Esencialmente, es un arreglo multidimensional. Los tensores son similares a los arreglos de NumPy, pero con la capacidad adicional de poder ser utilizados en una GPU para acelerar el cálculo.

- ▶ **Escalar:** Tensor de 0 dimensiones.
- ▶ **Vector:** Tensor de 1 dimensión.
- ▶ **Matriz:** Tensor de 2 dimensiones.
- ▶ Y así sucesivamente...

Escalares (Tensores 0-D)

Un escalar es un único número. En PyTorch, se representa como un tensor sin dimensiones.

```
import torch

# Crear un escalar
escalar = torch.tensor(7)
print(escalar)

# Verificar dimensiones
print(escalar.ndim)

# Obtener el valor como item de Python
print(escalar.item())
```

```
>> tensor(7)
>> 0
>> 7
```

Vectores (Tensores 1-D)

Un vector es un arreglo de números. Se representa como un tensor de una dimensión.

```
import torch

# Crear un vector
vector = torch.tensor([1, 2, 3, 4])
print(vector)

# Verificar dimensiones
print(vector.ndim)

# Verificar la forma (shape)
print(vector.shape)
```

```
>> tensor([1, 2, 3, 4])
>> 1
>> torch.Size([4])
```

Matrices (Tensores 2-D)

Una matriz es un arreglo bidimensional. En PyTorch, es un tensor de dos dimensiones.

```
import torch

# Crear una matriz
matriz = torch.tensor([[1, 2, 3],
                      [4, 5, 6]])
print(matriz)

# Verificar dimensiones
print(matriz.ndim)

# Verificar la forma (filas, columnas)
print(matriz.shape)
```

```
>> tensor([[1, 2, 3],
           [4, 5, 6]])
>> 2
>> torch.Size([2, 3])
```

Tensores de mas Dimensiones (3D, 4D, etc.)

Se pueden crear tensores con cualquier número de dimensiones. Son útiles para representar datos complejos como imágenes a color (alto, ancho, canales de color).

```
# Crear un tensor de 3 dimensiones
tensor_3d = torch.tensor([[ [1, 2], [3, 4]],
                           [[5, 6], [7, 8]]])
print(tensor_3d)

# Verificar dimensiones
print(tensor_3d.ndim)

# Verificar la forma
print(tensor_3d.shape)
```

```
>> tensor([[ [1, 2], [3, 4]],
               [[5, 6], [7, 8]]])
>> 3
>> torch.Size([2, 2, 2])
```

Aceleración con GPU

PyTorch permite mover tensores a la GPU para realizar cálculos masivamente paralelos, lo cual es crucial para el entrenamiento de redes neuronales.

```
# Verificar si la GPU está disponible
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Usando el dispositivo: {device}")

# Crear un tensor en la CPU
tensor_cpu = torch.tensor([[1, 2], [3, 4]])

# Mover el tensor a la GPU
tensor_gpu = tensor_cpu.to(device)

print(tensor_cpu.device)
print(tensor_gpu.device)
```

```
>> Usando el dispositivo: cuda
>> cpu
>> cuda:0
```

Creación de Parámetros Entrenables

Los parámetros de un modelo neuronal (pesos y bias) deben ser tensores que requieran cálculo de gradientes para poder ser optimizados. Esto se logra con 'requires_grad=True'.

```
# Crear un tensor que requiere gradientes
pesos = torch.randn(3, 4, requires_grad=True)
sesgo = torch.randn(4, requires_grad=True)

print(pesos.requires_grad)
print(sesgo.requires_grad)
```

Cualquier operación con ellos seguirá el rastro para el cálculo de gradientes (autograd).

Creación de Parámetros Entrenables

- ▶ Vamos a ver un ejemplo que calcule el gradiente de una función usando Pytorch.

```
import torch  
x = torch.tensor([[2., -1.], [1., 1.]], requires_grad=True)  
print(x)
```

- ▶ Luego, definimos la función de salida, que en este caso es la suma de los cuadrados de la entrada:

$$y = \sum_{i=1}^4 x_i^2$$

que con código queda

```
y = x.pow(2).sum()
```

Sabemos que la derivada de y va a ser $2x$. Vamos a verificarlo con las funciones de Pytorch

Creación de Parámetros Entrenables

- ▶ El gradiente de y se calcula utilizando el método `backward()`. El cálculo del gradiente - un cambio en y por un pequeño cambio en x (entrada con gradiente habilitado) es:

```
y.backward()
```

- ▶ Podemos ahora obtener el gradiente de y con respecto a x como sigue:

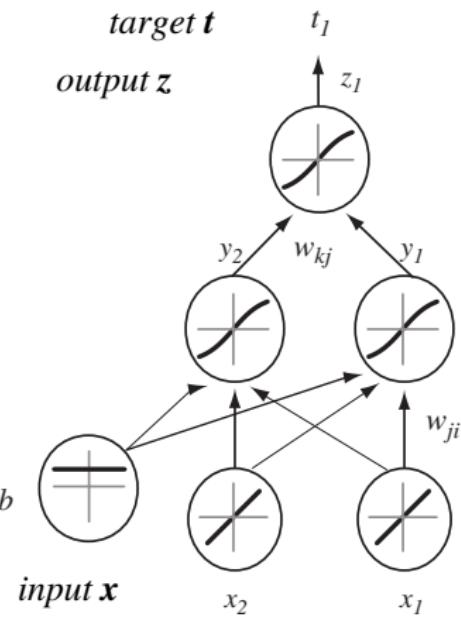
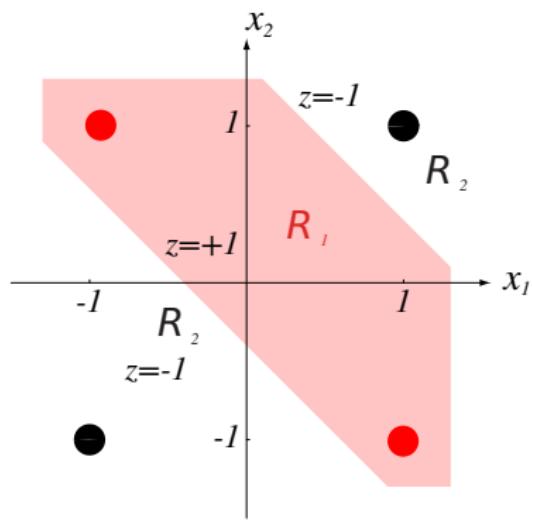
```
x.grad
```

lo que da como salida

```
>> tensor([[4., -2.],  
           [2., 2.]])
```

Notar que los gradientes obtenidos corresponden al valor esperado (dos veces el valor de x).

Modelos de Redes Neuronales con 'nn.Module'



Modelos de Redes Neuronales con 'nn.Module'

En PyTorch, los modelos de redes neuronales se construyen como clases que heredan de 'torch.nn.Module'. Las capas del modelo se definen en el constructor '__init__' y la lógica de propagación hacia adelante (forward pass) en el método 'forward'.

```
import torch.nn as nn

class MiRed(nn.Module):
    def __init__(self):
        super(MiRed, self).__init__()
        # Definir las capas
        self.capa1 = nn.Linear(in_features=2, out_features=2)
        self.capa2 = nn.Linear(in_features=2, out_features=1)

    def forward(self, x):
        # Definir el flujo de los datos
        y = torch.tanh(self.capa1(x))
        z = self.capa2(y)
        return torch.tanh(z)
```

Implementación de una Red

Una vez definida la clase del modelo, se puede crear una instancia y pasarle datos.

```
# Crear una instancia del modelo
modelo = MiRed()
print(modelo)

# Crear datos de entrada de ejemplo
# (1 muestra, 2 características)
datos_entrada = torch.randn(1, 2)

# Realizar una pasada hacia adelante
prediccion = modelo(datos_entrada)
print(f"Salida del modelo: {prediccion}")
```

```
>> Salida del modelo: tensor([-0.7719]), grad_fn=<TanhBackward0>
```

Función de Pérdida (Loss Function)

La función de pérdida mide qué tan lejos está la predicción del modelo del valor real. PyTorch ofrece varias funciones de pérdida predefinidas en 'torch.nn'.

```
import torch.nn as nn

# Ejemplo para regresión: Error Cuadrático Medio
funcion_perdida = nn.MSELoss()

# Ejemplo para clasificación: Entropía Cruzada
# funcion_perdida = nn.CrossEntropyLoss()

# Datos de ejemplo
objetivo = torch.tensor([10.0])
prediccion = torch.tensor([10.5])

# Calcular la pérdida
perdida = funcion_perdida(prediccion, objetivo)
print(f"Pérdida calculada: {perdida}")
```

>> Perdida calculada: 0.25

El optimizador actualiza los parámetros del modelo para minimizar la función de pérdida. Se utiliza el algoritmo de descenso de gradiente y sus variantes.

Proceso de Entrenamiento

1. '**optimizer.zero_grad()**': Limpiar los gradientes de la iteración anterior.
2. '**loss.backward()**': Calcular los gradientes de la pérdida con respecto a los parámetros (backpropagation).
3. '**optimizer.step()**': Actualizar los parámetros del modelo usando los gradientes calculados.

Optimización de la Red

```
import torch.optim as optim

# Instanciar el modelo y el optimizador
modelo = MiRed()
optimizador = optim.SGD(modelo.parameters(), lr=0.01)

# ... dentro de un bucle de entrenamiento ...
# optimizador.zero_grad()
# perdida.backward()
# optimizador.step()
```

Lectura de Datos con 'Dataset' y 'DataLoader'

PyTorch proporciona dos clases para manejar datos de manera eficiente:

- ▶ ‘**Dataset**’: Almacena las muestras y sus etiquetas correspondientes.
- ▶ ‘**DataLoader**’: Envuelve un ‘Dataset’ para proporcionar un iterable que facilita el manejo de lotes (batches), el muestreo y la carga de datos en paralelo.

Lectura de Datos con 'Dataset' y 'DataLoader'

```
from torch.utils.data import Dataset, DataLoader

# 1. Crear un Dataset personalizado
class MiDataset(Dataset):
    def __init__(self, datos, etiquetas):
        self.datos = datos
        self.etiquetas = etiquetas
    def __len__(self):
        return len(self.datos)
    def __getitem__(self, idx):
        return self.datos[idx], self.etiquetas[idx]
# 2. Instanciar Dataset y DataLoader
# (Suponiendo que 'X_train', 'y_train' existen)
# dataset_entrenamiento = MiDataset(X_train, y_train)
# dataloader = DataLoader(dataset_entrenamiento,
#                         batch_size=32, shuffle=True)
# 3. Iterar sobre los datos en el bucle de entrenamiento
# for datos_lote, etiquetas_lote in dataloader:
#     # ... lógica de entrenamiento ...
```

Resumiendo

Vamos a armar un ejemplo de un código completo de definición y entrenamiento de una red neuronal (MiRed).

```
from torch.utils.data import Dataset, DataLoader,  
                           SubsetRandomSampler  
import torch.optim as optim  
  
# Instanciar el modelo, loss, optimizador y dataset  
modelo = MiRed()  
loss_fn = nn.MSELoss()  
optimizador = optim.SGD(modelo.parameters(), lr=0.01)  
mdataset = Dataset(X_train, y_train)
```

Resumiendo

```
# dividir el dataset en entrenamiento y test
x = np.array(range(len(mdataset)))
idx_test = x[0:int(len(mdataset)*0.3)]
idx_train = x[int(len(mdataset)*0.3):]
# Instanciar los samplers correspondientes
train_subampler = SubsetRandomSampler(idx_train)
test_subampler = SubsetRandomSampler(idx_test)
# Definir los dataloaders para entrenamiento y test
trainloader = DataLoader(mdataset, batch_size=25,
                        sampler=train_subampler)
testloader = DataLoader(mdataset, batch_size=25,
                        shuffle=False,
                        sampler=test_subampler)
```

Resumiendo

```
epochs = 50
for t in range(epochs):
    print(f"Epoch {t+1}\n-----")
    model = train(trainloader, model, loss_fn, optimizer, device)
    test(testloader, model, loss_fn, device)
    torch.save(model.state_dict(), 'modelo_weights.pth')
print("Done!")
```

Resumiendo

```
def train(dataloader, model, loss_fn, optimizer, device):
    size = len(dataloader.dataset)
    model.train()
    for batch, (datos, etiquetas) in enumerate(dataloader):
        datos, etiqueta = datos.to(device),
                                etiqueta.to(device)
        # Compute prediction error
        pred = model(datos)
        loss = loss_fn(pred, y, X_mk)
        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
    return model
```

Resumiendo

```
def test(dataloader, model, loss_fn, device):
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for datos, etiquetas in dataloader:
            datos, etiquetas = datos.to(device),
                                etiquetas.to(device)
            pred = model(datos)
            test_loss += loss_fn(pred, etiquetas).item()
    test_loss /= num_batches
    print(f"Test Error: \n Avg loss: {test_loss:>8f} \n")
```

Autoencoders

Salidas categóricas vs Salidas de Datos

Hasta ahora veníamos viendo sistemas que tienen por finalidad realizar una estimación que tiene una instancia sobre la probabilidad de pertenencia a una categoría o clase.

En esta clase, veremos arquitecturas con otro tipo de salidas:

- ▶ Autoencoder: en este caso se busca obtener una función tal que $f : \mathcal{X} \rightarrow \mathcal{X}$ consiguiendo que $f(\mathbf{x}) = \hat{\mathbf{x}}$ y $\hat{\mathbf{x}} \approx \mathbf{x}$.

Salidas categóricas vs Salidas de Datos

Hasta ahora veníamos viendo sistemas que tienen por finalidad realizar una estimación que tiene una instancia sobre la probabilidad de pertenencia a una categoría o clase.

En esta clase, veremos arquitecturas con otro tipo de salidas:

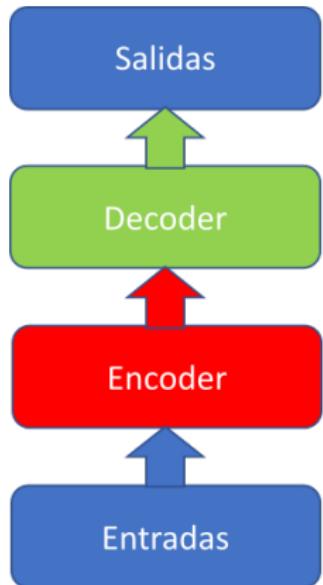
- ▶ Autoencoder: en este caso se busca obtener una función tal que $f : \mathcal{X} \rightarrow \mathcal{X}$ consiguiendo que $f(\mathbf{x}) = \hat{\mathbf{x}}$ y $\hat{\mathbf{x}} \approx \mathbf{x}$.
- ▶ Segmentación: U-Net es una arquitectura de red que produce una salida categórica a nivel del pixel de la imagen de entrada. En este caso el espacio de pares se representa por $(\mathbf{X}_1, \mathbf{Y}_1), \dots, (\mathbf{X}_m, \mathbf{Y}_m)$, con $\mathbf{X}_i \in \mathbb{R}^{H \times W \times 3}$ y $\mathbf{Y}_i \in \mathbb{Z}^{I \times J \times C}$. C puede ser la cantidad de clases o categorías a las que puede ser clasificado un pixel.

Definición

Los autoencoders son sistemas que están especializados en copiar la entrada de datos.

Están compuestos habitualmente por dos partes:

- ▶ Encoder: convierte la entrada en una representación interna,
- ▶ Decoder: reconstruye desde la representación interna una salida del mismo tamaño que la entrada.

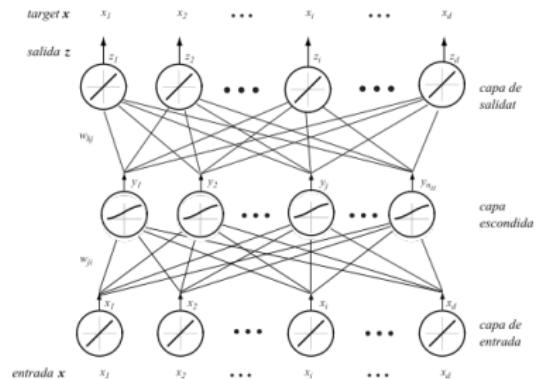


Autoencoders

Arquitectura

En una red neuronal, podemos definir:

- ▶ los datos de entrada como el vector $\mathbf{x} = x_1, \dots, x_d$ de tamaño d
- ▶ una capa encoder: compuesta de n_H neuronas ($n_H < d$)
- ▶ una capa decoder: con d neuronas de salidas, y sabiendo que cada $z_i = \hat{x}_i$ tiene como target x_i .



Arquitectura

Los autoencoders son redes MLP convencionales pero:

- ▶ el número de salidas es el mismo de la entrada
- ▶ poseen una función de pérdida que penaliza cuando la salida difiere de la entrada

Dado que el número de neuronas de la capa escondida es menor que d , la red está forzada a aprender relaciones entre los píxeles (features) que mejor representen la entrada y que permitan una reconstrucción fidedigna.

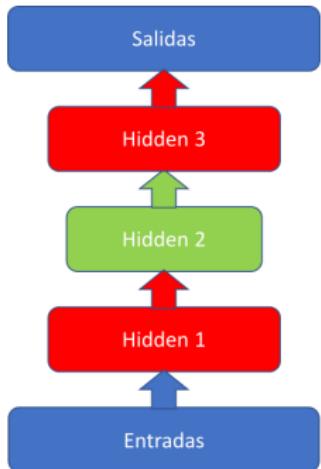
Arquitectura

Las propiedades de los autoencoders consisten en:

- ▶ reducción de la dimensionalidad del espacio de entrada (comportamiento similar a PCA),
- ▶ son capaces de extraer potentes features,
- ▶ pueden generar aleatoriamente salidas similares a la entrada (modelos de tipo generativos).

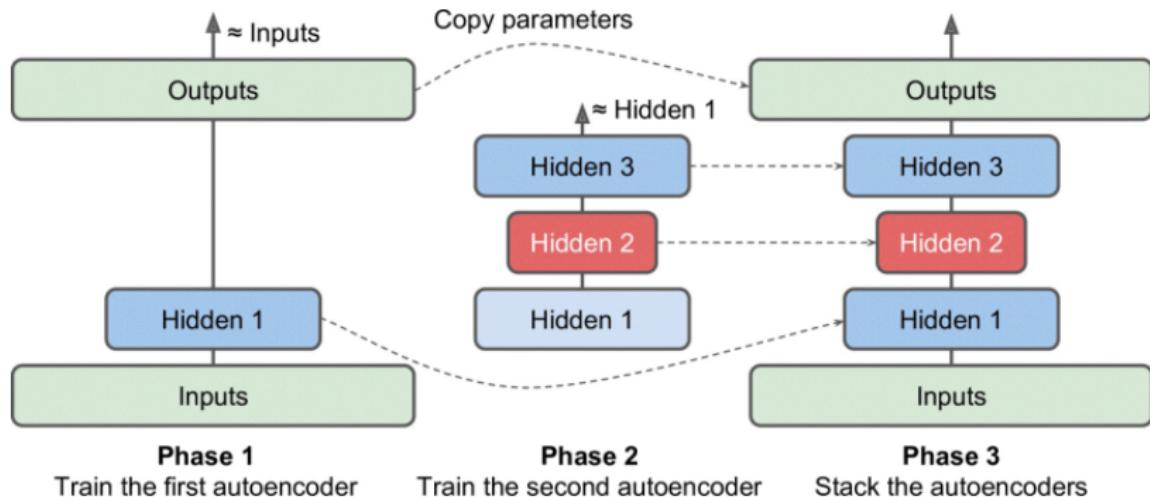
Stacked Autoencoders

- ▶ Los *stacked* son la versión profunda de los autoencoders,
- ▶ Capas adicionales permitirían el aprendizaje de features más complejos.
- ▶ Generalmente es una estructura simétrica ($|H1| = |H3|$).



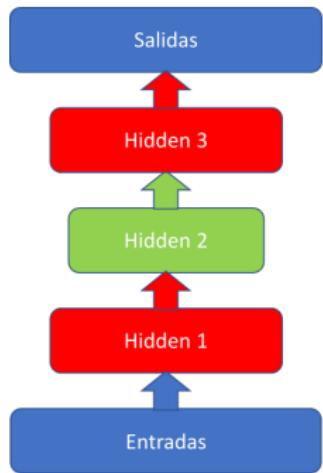
Stacked Autoencoders - Aprendizaje

El aprendizaje puede acelerarse (sobre todo ante muchas capas) entrenando una capa individualmente cada vez.



Stacked Autoencoders - Tying weights

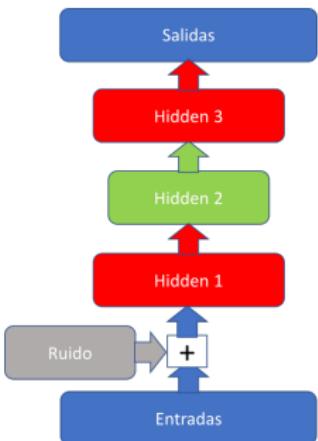
- ▶ Esta modificación consiste en utilizar los mismos pesos del *decoder* con el *encoder*.
- ▶ En el ejemplo: $\mathbf{W}_{H1} = \mathbf{W}_{H3}$,
- ▶ Gracias a esto, la cantidad de parámetros de aprendizaje baja drásticamente.



Autoencoders

Stacked Autoencoders

- ▶ Los autoencoders podrían converger a una función identidad, copiando fácilmente la entrada.
- ▶ Para evitar esto, se puede adicionar ruido antes de la entrada a la red, incorporando diversidad y evitando el overfitting.



Segmentación

Segmentación

Introducción

El Pascal Context-Challenge es un desafío abierto que propone resolver la problemática de la segmentación³. El desafío consiste en presentar imágenes y que el sistema sea capaz de dar una respuesta pixel a pixel sobre la clase a la que pertenece.



Formalmente, el espacio de datos es:

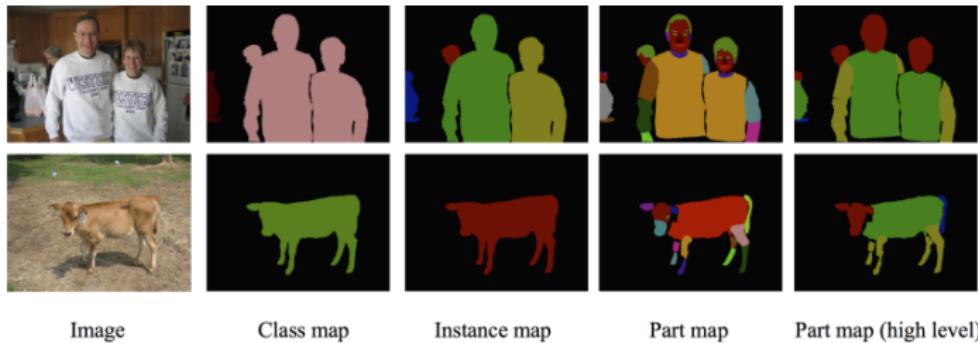
$$(\mathbf{X}_1, \mathbf{Y}_1), \dots, (\mathbf{X}_m, \mathbf{Y}_m) \in \mathcal{S}$$

con $\mathbf{X} \in \mathbb{Z}^{n \times m \times 3}$ imágenes en colores, $\mathbf{Y} \in \mathbb{Z}^{h \times w}$ un mapa de labels.

³<https://cs.stanford.edu/~roozbeh/pascal-context/>

Segmentación

Introducción



- ▶ Reconocer una clase en la imagen,
- ▶ Discriminar las distintas instancias de una clase,
- ▶ Identificar contenido de todas las partes.

U-Net

En 2015 Ronneberger et al.⁴ proponen una arquitectura convolucional especialmente adaptada para la tarea de segmentación.



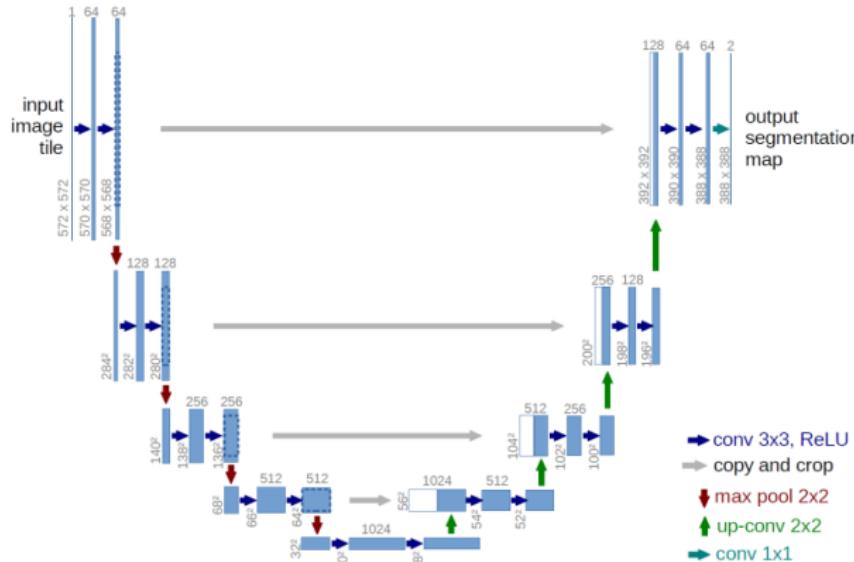
La arquitectura reduce la información visual en un vector de características reduciendo las dimensiones con un *downsampling* por pooling. Luego, este vector se utiliza de generador para, haciendo *upsampling*, generar una matriz de píxeles con información de clase asociada.

⁴O. Ronneberger, P. Fischer, and T. Brox. "U-Net: Convolutional networks for biomedical image segmentation." International Conference on Medical image computing and computer-assisted intervention. Springer, Cham, 2015.

Segmentación

U-Net

En 2015 Ronneberger et al.⁵ proponen una arquitectura convolucional especialmente adaptada para la tarea de segmentación.



La forma de la arquitectura termina siendo una U.

⁵O. Ronneberger, P. Fischer, and T. Brox. "U-Net: Convolutional networks for biomedical image segmentation." International Conference on Medical image computing and computer-assisted intervention. Springer, Cham, 2015.

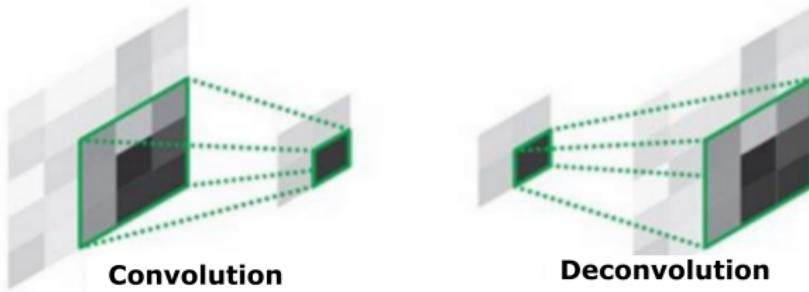
U-Net

U-Net posee diversas características y ventajas:

- ▶ Se puede entrenar con un número limitado de instancias,
- ▶ No posee capas *fully connected*,
- ▶ Se puede usar *data augmentation* a partir de simples deformaciones elásticas,
- ▶ Se aplica una función de loss especial para tratar el caso de clases que se "tocan", con una función de borde.

Deconvolución

La necesidad de la deconvolución surge de precisar realizar la operación inversa a la convolución tradicional, en el sentido de usar una transformación de una capa o un mapa de features a un espacio dimensional de mayor dimensión.



Deconvolución

La necesidad de la deconvolución surge de precisar realizar la operación inversa a la convolución tradicional, en el sentido de usar una transformación de una capa o un mapa de features a un espacio dimensional de mayor dimensión.

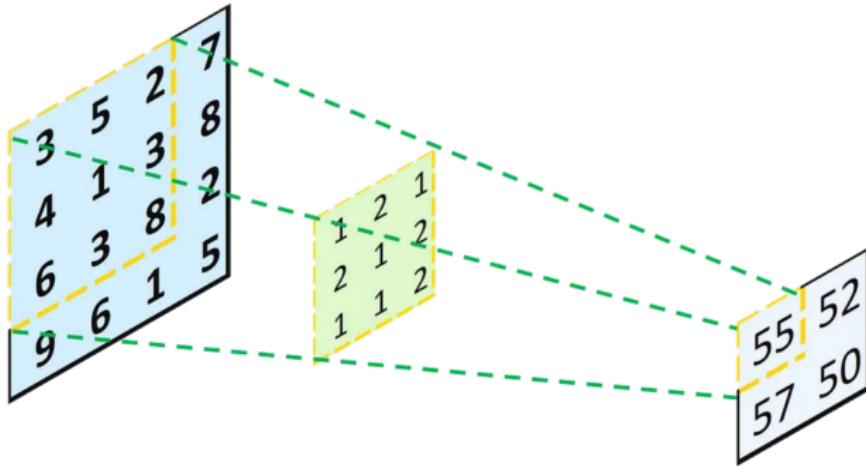
En este caso tenemos una entrada de tamaño 4×4 , un kernel de 3×3 , y la salida de la convolución de tamaño 2×2 .

Deconvolución

Si la entrada y la salida se aplanan en vectores de izquierda a derecha y de arriba a abajo, la convolución podría representarse como una matriz dispersa \mathbf{C} donde los elementos distintos de cero son los elementos $w_{i,j}$ del núcleo (siendo i y j la fila y la columna del kernel respectivamente):

$$\begin{pmatrix} w_{0,0} & 0 & 0 & 0 \\ w_{0,1} & w_{0,0} & 0 & 0 \\ w_{0,2} & w_{0,1} & 0 & 0 \\ 0 & w_{0,2} & 0 & 0 \\ w_{1,0} & 0 & w_{0,0} & 0 \\ w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\ w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\ 0 & w_{1,2} & 0 & w_{0,2} \\ w_{2,0} & 0 & w_{1,0} & 0 \\ w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\ w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\ 0 & w_{2,2} & 0 & w_{1,2} \\ 0 & 0 & w_{2,0} & 0 \\ 0 & 0 & w_{2,1} & w_{2,0} \\ 0 & 0 & w_{2,2} & w_{2,1} \\ 0 & 0 & 0 & w_{2,2} \end{pmatrix}^T$$

Deconvolución



Deconvolución

$$\begin{matrix} 1 & 2 & 1 & 0 & 2 & 1 & 2 & 0 & 1 & 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 2 & 1 & 2 & 0 & 1 & 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 & 2 & 1 & 2 & 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 & 2 & 1 & 2 & 0 & 1 & 1 & 2 \end{matrix} \times \begin{matrix} 3 \\ 5 \\ 2 \\ 7 \\ 4 \\ 1 \\ 3 \\ 8 \\ 6 \\ 3 \\ 8 \\ 2 \\ 9 \end{matrix} = \begin{matrix} 55 \\ 52 \\ 57 \\ 50 \end{matrix} \rightarrow \begin{matrix} 55 & 52 \\ 57 & 50 \end{matrix}$$

Deconvolución

1	0	0	0
2	1	0	0
1	2	0	0
0	1	0	0
2	0	1	0
1	2	2	1
2	1	1	2
0	2	0	1
1	0	2	0
1	1	1	2
2	1	2	1
0	2	0	2
0	0	1	0
0	0	1	1
0	0	2	1
0	0	0	2

X

55
52
57
50

=

55	0	0	0
110	52	0	0
55	104	0	0
0	52	0	0
110	0	57	0

+

0	104	114	50
110	52	57	100
55	0	114	0
55	52	57	100
110	52	114	50

+

0	0	57	50
0	0	57	50
0	0	114	50
0	0	0	100
0	0	114	0

=

55	162	159	52
167	323	319	154
169	264	326	204
57	107	164	100
204	57	104	164
264	326	204	100
319	154	169	264
323	154	169	264
154	169	264	326
169	264	326	204
154	169	264	326
167	323	319	154
319	154	169	264
154	169	264	326
204	57	104	164
57	107	164	100



U-Net

El entrenamiento de la red se lleva a cabo por:

- ▶ Stochastic Gradient Descent, y un *momentum* = 0.99,
- ▶ La función de costo se calcula a partir de un soft-max:

$$p_k(i, j) = \frac{e^{a_k(i, j)}}{\sum_{k'}^K e^{a_{k'}(i, j)}}$$

donde $a_k(i, j)$ es la activación de la posición (i, j) , K el número de clases.

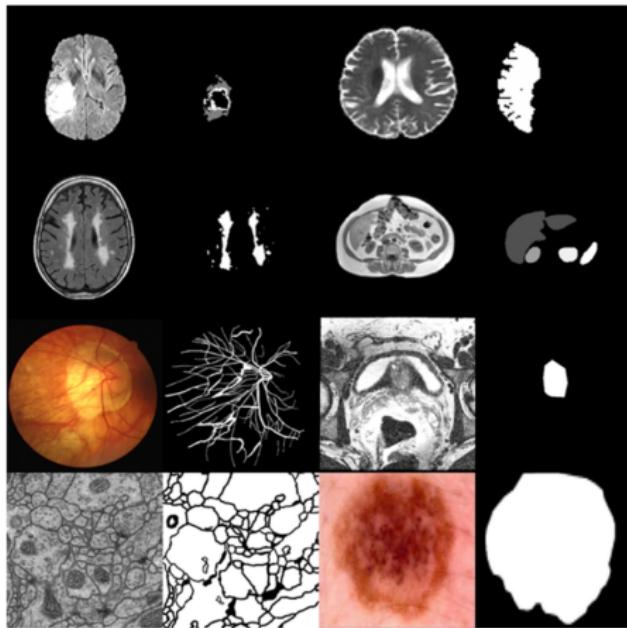
- ▶ La función de cross-entropy penaliza la desviación de $p_\ell(i, j)$ del valor 1 como:

$$E = \sum_{(i, j) \in \Omega} w(i, j) \log(p_\ell(i, j))$$

Segmentación

U-Net

Las U-Net son ampliamente utilizadas para la segmentación de imágenes médicas.



Segmentación

U-Net

Las U-Net son ampliamente utilizadas para la segmentación de imágenes médicas.

