Tema 06. Deep Learning

Autor: Ismael Sagredo Olivenza

7.1 Introducción DeepLearning

El algoritmo de aprendizaje basado en gradiente tiene un problema fundamental y es que cada una de las capas aprende a diferentes velocidades. Las capas mas cercanas a la salida les afecta más que a las primeras. De esta forma, necesitamos otra forma de funcionar para permitir aumentar la capacidad de representación de la red sin que el gradiente del error se vuelva inestable.

Se ha demostrado que una red con una sola capa oculta puede modelar funciones muy complejas, siempre que tenga suficientes neuronas. Durante mucho tiempo, estos convenció a los investigadores de que no había necesidad de redes más profundas.

Pero pasaron por alto el hecho de que las redes profundas tienen una eficiencia de parámetros mucho más alta que las superficiales.

Es decir, **pueden modelar funciones complejas utilizando exponencialmente menos neuronas que redes poco profundas**.

En cuanto a las capas ocultas, una práctica común es dimensionarlas para formar un **embudo**, es decir, la capa i tendrá (bastantes) más neuronas que la capa siguiente i + 1. Esta aproximación se basa en que muchas características de bajo nivel pueden unirse en muchas menos características de alto nivel. **Las capas iniciales trabajan con muchas características de bajo nivel**, mientras que las capas ocultas cercanas a la capa de salida, **trabajan con menos características de más alto nivel.**

Las funciones de activación, se suele usar la función de activación **ReLU** en las capas ocultas (o una de sus variantes). Esta función es un poco más rápida de computar que otras funciones de activación, y facilita la fase de entrenamiento evitando la saturación de las neuronas de las capas ocultas.

Para la capa de salida, la función de activación **softmax** es generalmente una buena opción para tareas de clasificación **cuando las clases son mutuamente excluyentes**. Cuando no son mutuamente excluyentes (o cuando solo hay dos clases), generalmente se prefiere **la función logística**.

Para las tareas de regresión, se suelen emplear funciones lineales de activación.

7.1.1 Softmax

Son neuronas que normalmente se colocan en la capa de salida y que recogen las entradas de las red de una forma similar a como hace la función sigmoide, sumando ponderadamente los valores de la capa previa a la capa actual. Pero en vez de aplicar la función sigmoide, se aplica la siguiente función:

$$y_j = \frac{e_j^Z}{\sum_k (e_k^Z)}$$

Es decir, que la suma de todas las salidas suma 1 porque cada una de los valores producidos se divide entre la suma de todos ellos.

Siendo z_j:

$$z_j^C = \sum_k W_{i,k} \cdot y_k^{C-1} + u_j^C$$

Donde u_j^C es el umbral de cada neurona de salida.

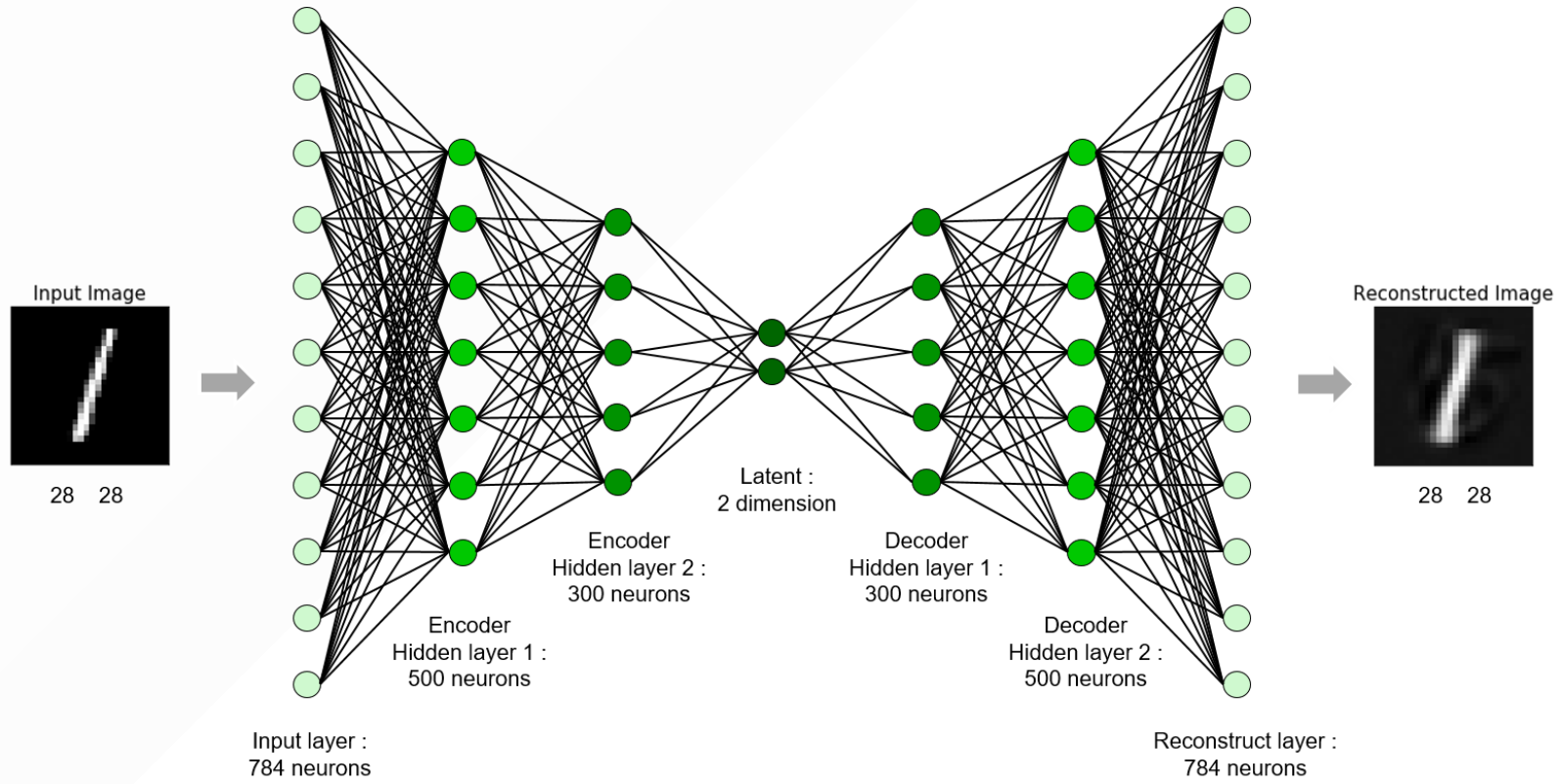
Esta función permite interpretar la salida como una probabilidad. Cada una de las salidas de la red es una clase y su valor la probabilidad de esa sea la clase de los datos.

7.1.2 Codigo en Python dela función softmax

```
def SoftMax(z):
        z_exp = [math.exp(i) for i in z] # e^z
        sum_z_exp = sum(z_exp)
        softmax = [round(i / sum_z_exp, 3) for i in z_exp]
        return softmax
```

7.2 Autoencoders

Son un tipo especial de redes neuronales que funcionan intentando reproducir los datos de entrada en la salida de la red.



Aunque aparentemente no tiene mucha utilidad, si configuramos la red en forma de doble embudo, y entrenamos la red, podemos dividir la red en dos por la parte estrecha. Asi tenemos una red que comprime la información de entrada en unos pocos valores.

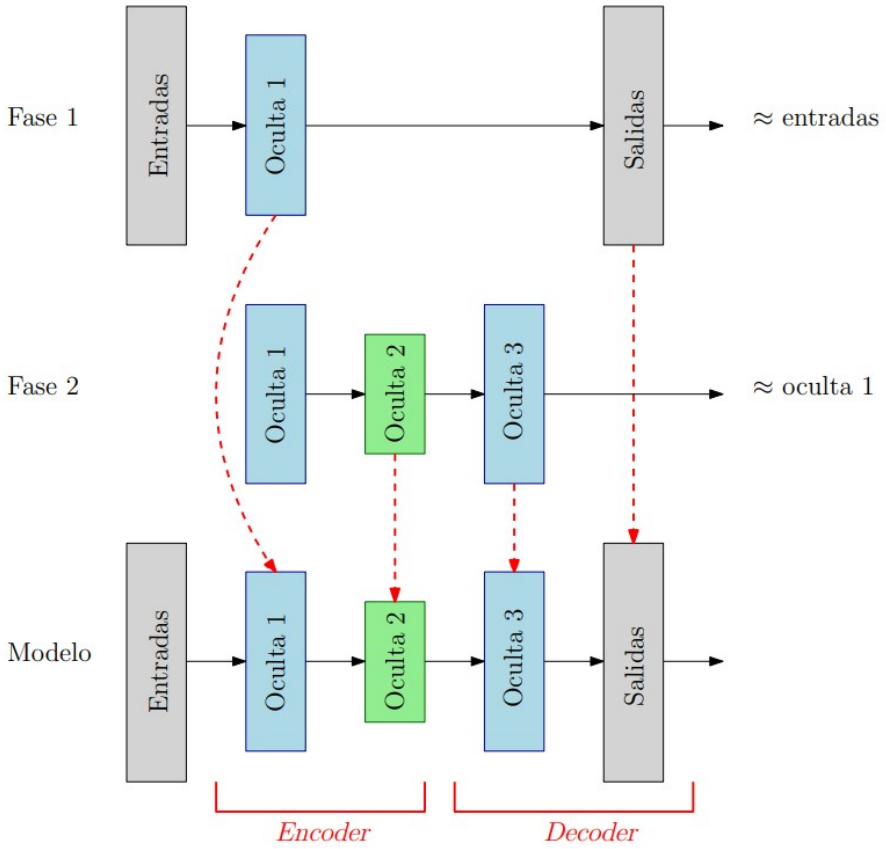
Y la segunda mitad tenemos un descompresor. Pero más alla de su posible utilización como compresión, lo interesante es que este tipo de redes permite extraer las caracteristicas más relevantes de una red usando aprendizaje no supervisado.

No suelen utilizar neuronas Softmax en la capa de salida si no sigmoidal o ReLu.

Los autoencoders pueden tener multiples capas ocultas (auto encoders apilados o stacked)

El entrenamiento de las redes se realiza de forma análoga a las tradicionales.

Si hay muchas capas, se puede entrenar las capas poco a poco (capa a capa) para simplificar el entrenamiento.



```
import torch
from torchvision import datasets
from torchvision import transforms
import matplotlib.pyplot as plt

# Transforms images to a PyTorch Tensor
tensor_transform = transforms.ToTensor()

# Download the MNIST Dataset
dataset = datasets.MNIST(root = "./data",
                         train = True,
                         download = True,
                         transform = tensor_transform)

# DataLoader is used to load the dataset
# for training
loader = torch.utils.data.DataLoader(dataset = dataset,
                                     batch_size = 32,
                                     shuffle = True)
```

```
class AE(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.encoder = torch.nn.Sequential(
            torch.nn.Linear(28 * 28, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 36),
            torch.nn.ReLU(),
            torch.nn.Linear(36, 18),
            torch.nn.ReLU(),
            torch.nn.Linear(18, 9)
        )
```

```
        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(9, 18),
            torch.nn.ReLU(),
            torch.nn.Linear(18, 36),
            torch.nn.ReLU(),
            torch.nn.Linear(36, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 28 * 28),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

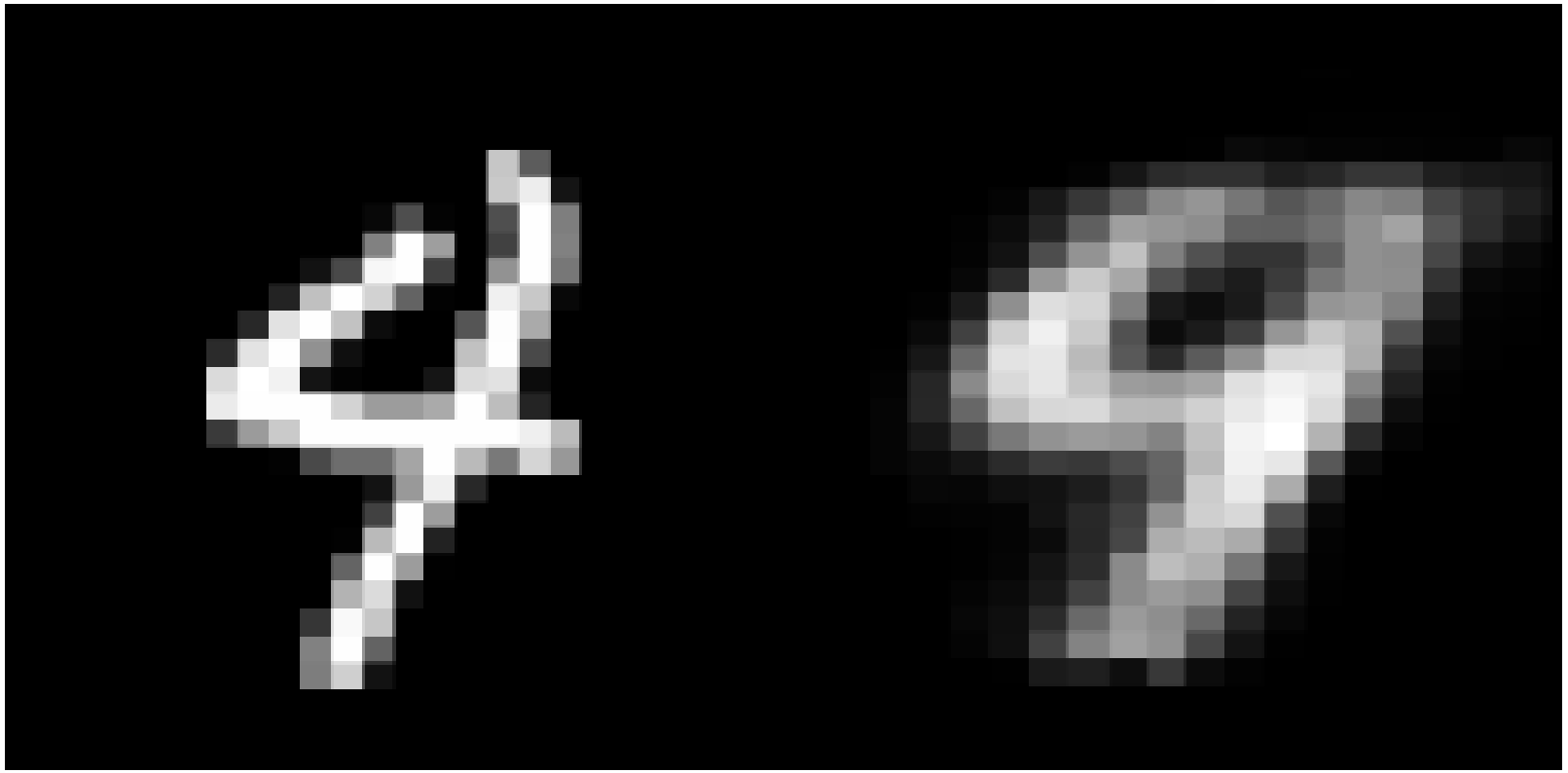
- transforms.ToTensor() : convierte una imagen en formato de tensor. Un tensor es un vector o una matriz.
- Linear: Aplica una transformación lineal: $y = xA^T + b$
- ReLU: aplica una rectificación ReLU (x) = max(0,x)
- Sigmoid_ Aplica una capa con función de activación sigmoide (como el perceptrón multicapa) para que devuelva el valor 0 o 1 (nótese que no hay neuronas en la capa, se asume que sólo hay una)

USamos un optimizador ADAM para entrenar la red y MSE como función de coste o LOSS.

```
# Model Initialization
model = AE()

# Validation using MSE Loss function
loss_function = torch.nn.MSELoss()

# Using an Adam Optimizer with lr = 0.1
optimizer = torch.optim.Adam(model.parameters(),
                             lr = 1e-1,
                             weight_decay = 1e-8)
```

```
epochs = 20
outputs = []
losses = []
for epoch in range(epochs):
    for (image, _) in loader:

      # Reshaping the image to (-1, 784)
      image = image.reshape(-1, 28*28)

      # Output of Autoencoder
      reconstructed = model(image)

      # Calculating the loss function
      loss = loss_function(reconstructed, image)

      # The gradients are set to zero,
      # the gradient is computed and stored.
      # .step() performs parameter update
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

      # Storing the losses in a list for plotting
      losses.append(loss)
    outputs.append((epochs, image, reconstructed))
```

Input a la izquierda e imagen reconstruida a la derecha.



7.3 Convolutional networks

Las redes convolucionales (LeCun, 1989) son un tipo especial de redes neuronales para procesar datos con tipología cuadriculada, como las imágenes.

El problema de las imágenes es que el tamaño de los datos es enorme. Si tenemos una imagen de de 1000x1000 pixeles a 3 canales por pixel tenemos un vector de entrada de 10^6 pixeles. El número de pesos de la red cuando bajemos la siguiente capa a pongamos 1000, esta capa de pesos estaría manejando $10^6 x 10^3$ pixeles o lo que es lo mismo 10^9 pesos para la primera capa. Entrenar esto sin que se produzca overfeating necesitaría de muchísimos datos.

7.3.1 Reducir las dimensiones de la imagen

Por le problema anterior necesitamos reducir la complejidad de la imagen, pero **conservando su información esencial**.

La **convolución** es una operación lineal sobre dos funciones f y g que producen una tercera función s que expresa como la forma de una es modificada por la otra y se suele denotar con un * o un circulo (o).

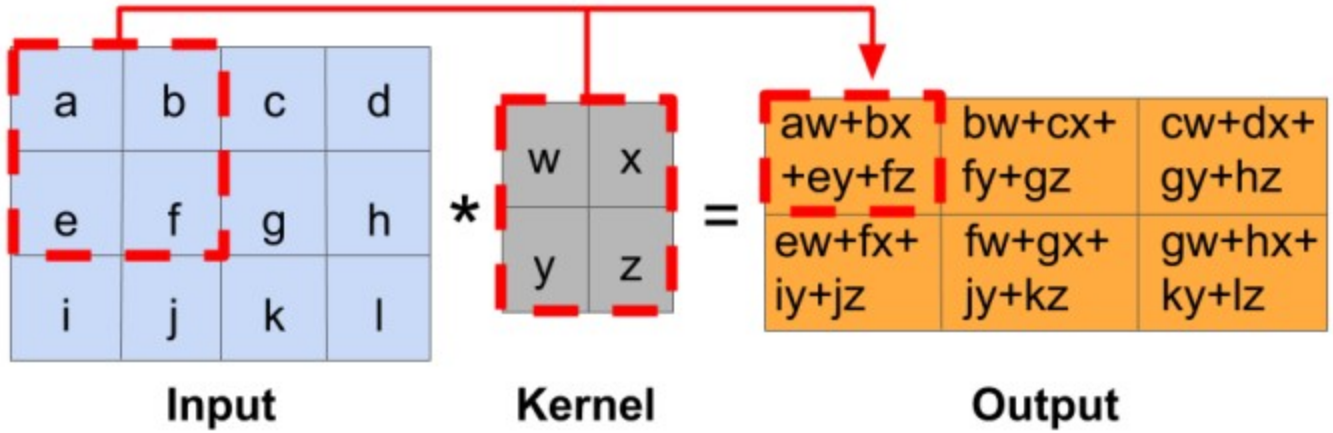
$$s(t) = (f * g)(t) = \int f(t)g(t - T)dT$$

f: es la entrada, S es el mapa de característica (feature map) y g: es el filtro o kernel.

En nuestro caso pasamos de la integral por ser datos discretos y lo convertimos en un sumatorio:

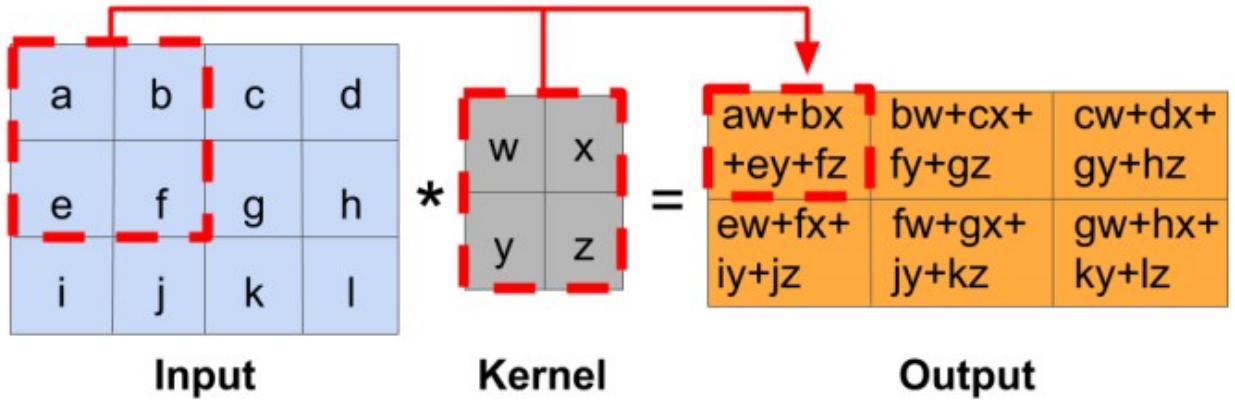
$$s(t) = (f * g)(t) = \sum_{T=-\infty}^{\infty} f(t)g(t-T)dT$$

g es un vector multidimensional de parámetros.



Hasta ahora conectábamos todas las neuronas de la entrada con la oculta. Lo cual multiplica exponencialmente los pesos. Las redes convolucionales hacen un mapeo de una capa a la siguiente, reduciendo el tamaño y sin interconectar todos los nodos

En el ejemplo anterior aplicamos un kernel 2x2 a una matriz de 4x4, convirtiendo dicha matriz en una matriz 3x3 que es una combinación lineal de los anteriores valores.

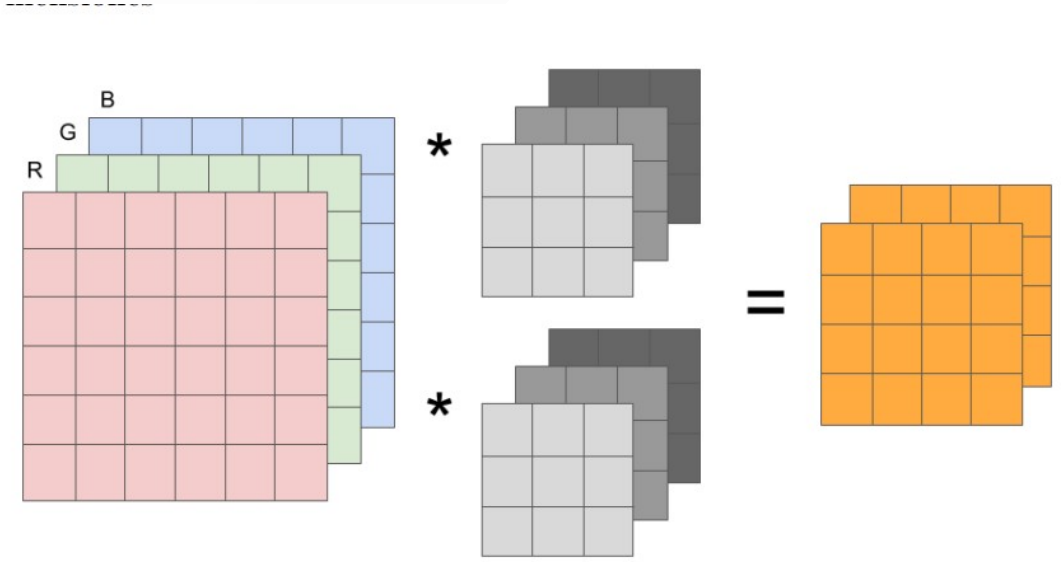


Cada capa oculta detecta la misma característica, pero esta detección la realiza por toda la capa previa. Los pesos compartidos (Share weights) y el sesgo compartido conforman el kernel.

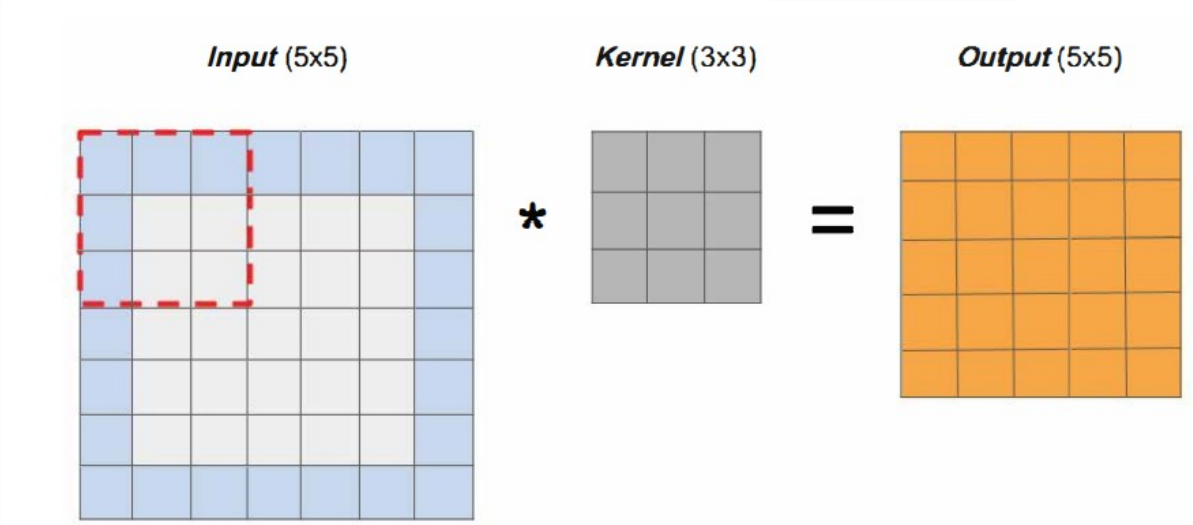
Las redes convolucionales deberán utilizar múltiples mapas de características según el número de patrones a detectar en los datos en entrada. La capa oculta por tanto puede estar compuesta por multiples mapas de características que se van extrayendo con cada uno de los kernels. Y estos además pueden ir extrayendo características cada vez de mayor nivel.

7.3.2 Partes de una CNN

- Kernel: Reduce la dimensionalidad de la imagen. Si la imagen tiene varios canales se peuden sumar los canales aplicando un kernel por cada canal (RGB) y combinándolos en uno solo. También podemos tener más de un kernel para extraer caracteristicas diferentes.



- Padding: añadir 0s para mantener la dimensionalidad de la matriz de salida (si asi lo queremos)



- Step : consiste en no aplicar la convolución a toda la matri, y saltarse pasos.

7.3.3 Otras capas

- Capa de agrupamiento (pooling): el max-pooling
  el más usado en las redes convolucionales, que simplemente selecciona el valor máximo del conjunto de valores de entrada
- Capa totalmente conectada (fully connected): para clasificar
- Capa ReLU: aplicar no linealidad
- Capa de dropout: desactiva un numero de entradas poniéndolas a 0 para evitar el sobreentrenamiento.

7.3.4 Aplicaciones

- Visión artificial en coches autónomos
- Reconocimiento facial
- Clasificación de imágenes
- Transferencia de estilos

7.4.5 Ejemplo en Pythorch

```
import torch.nn as nn
import torch.optim as optim
import torchvision

transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor()])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

batch_size = 32
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
```

```
class CIFAR10Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=(3,3), stride=1, padding=1)
        self.act1 = nn.ReLU()
        self.drop1 = nn.Dropout(0.3)

        self.conv2 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=1, padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=(2, 2))

        self.flat = nn.Flatten()

        self.fc3 = nn.Linear(8192, 512)
        self.act3 = nn.ReLU()
        self.drop3 = nn.Dropout(0.5)

        self.fc4 = nn.Linear(512, 10)
```

- Dropout: Durante el entrenamiento, poner a cero aleatoriamente algunos de los elementos del tensor de entrada con probabilidad p utilizando muestras de una distribución Bernoulli. Se ha demostrado eficaz para la regularización y prevenir la co-adaptación.

https://arxiv.org/abs/1207.0580

LA co-adaptación es un fenomeno que presentan las redes de neuronas que hace que partes de los pesos sea simétricos o realicen las mismas tareas que otros. Es decir, son redundantes. Dropout hace que algunos nodos se queden a 0, lo que limita le numero de nodos que se co-adaptan.

```
    def forward(self, x):
        # input 3x32x32, output 32x32x32
        x = self.act1(self.conv1(x))
        x = self.drop1(x)
        # input 32x32x32, output 32x32x32
        x = self.act2(self.conv2(x))
        # input 32x32x32, output 32x16x16
        x = self.pool2(x)
        # input 32x16x16, output 8192
        x = self.flat(x)
        # input 8192, output 512
        x = self.act3(self.fc3(x))
        x = self.drop3(x)
        # input 512, output 10
        x = self.fc4(x)
        return x
```

```
model = CIFAR10Model()
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
n_epochs = 20
for epoch in range(n_epochs):
    for inputs, labels in trainloader:
        # forward, backward, and then weight update
        y_pred = model(inputs)
        loss = loss_fn(y_pred, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    acc = 0
    count = 0
    for inputs, labels in testloader:
        y_pred = model(inputs)
        acc += (torch.argmax(y_pred, 1) == labels).float().sum()
        count += len(labels)
    acc /= count
    print("Epoch %d: model accuracy %.2f%%" % (epoch, acc*100))
```