REDES NEURONALES

CONVOLUTIVAS

MINERÍA DE DATOS

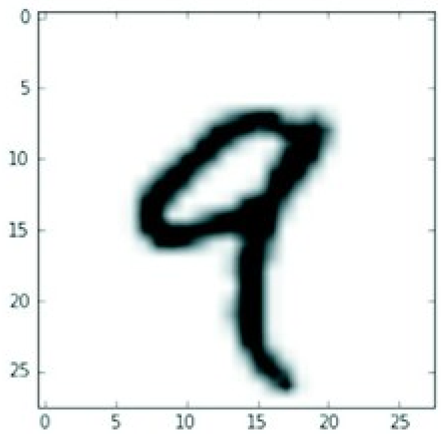
4º Curso. Grado en Ingeniería Informática
5º Curso. Doble Grado Informática/Estadística (INDAT)

UNIVERSIDAD DE VALLADOLID

Departamento de Informática (ATC, CCIA y LSI)

Tratamiento numérico de imágenes

MNIST



Matriz[28x28]:
Pixel → byte:
0 → oscuro
255 → blanco

Aplanado (flat):
Vector [768]

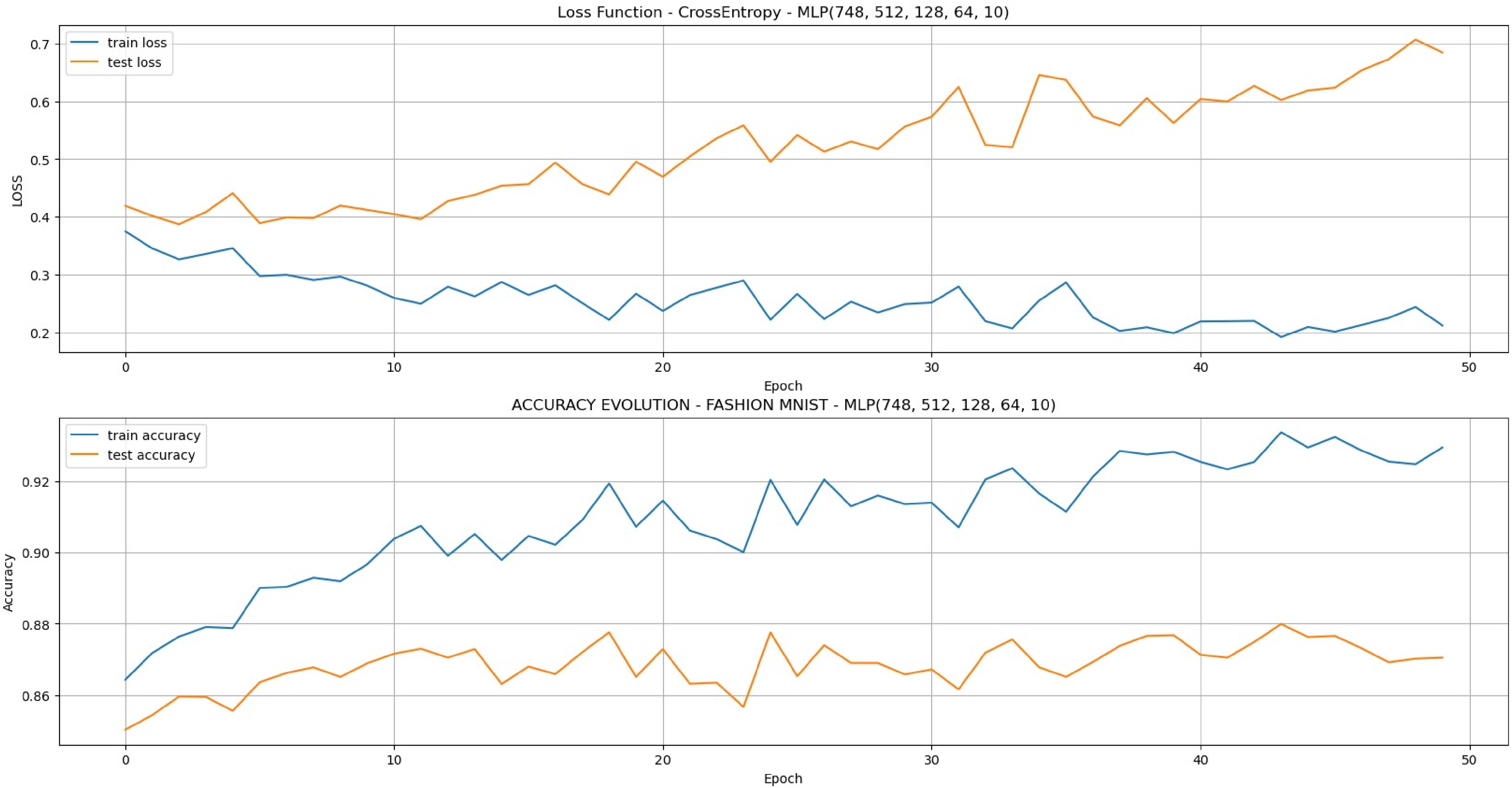
MLP
SVM
Rlog
.
.
.

- Ineficiencia al tener que convertir tipos **int→float**:
  Float32 (por defecto en Deep Learning) y Punto Fijo (no flotante)
- Pérdida de información al **aplanar**

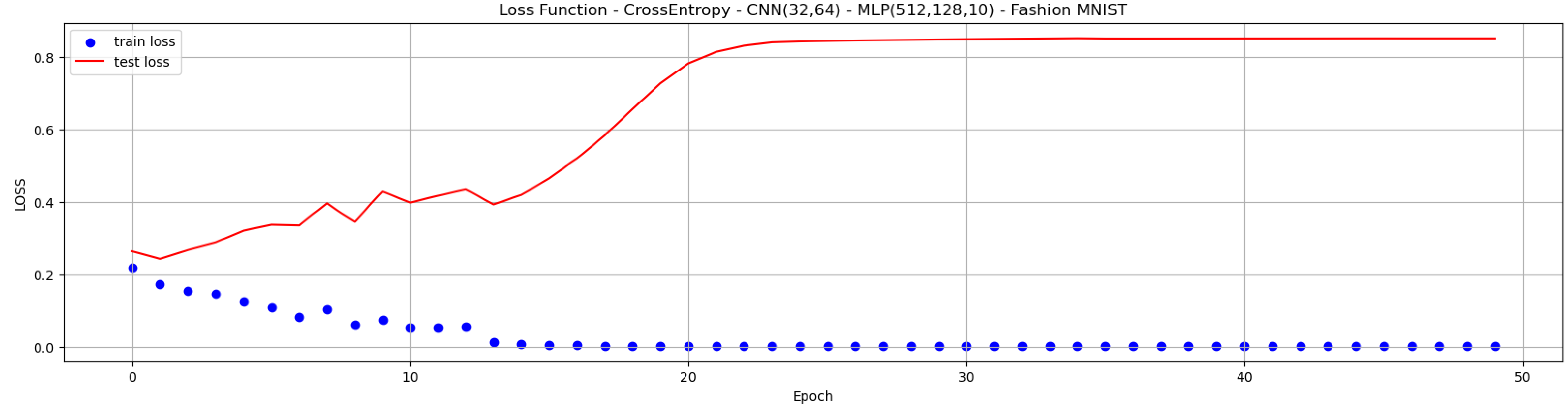
Redes convolutivas:

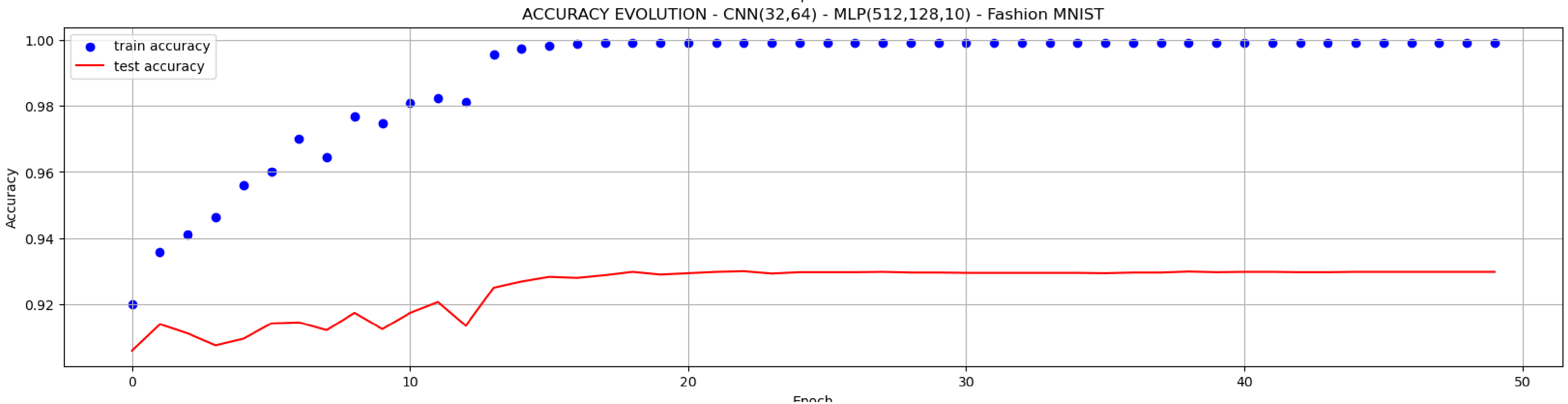
- Extraen características 2D
- Para ello aplican un núcleo, que es una matriz 2D

Fashion MNIST: MLP 512-128-64-10



CNN - VGG


Loss Function - CrossEntropy - CNN(32,64) - MLP(512,128,10) - Fashion MNIST
train loss
test loss
LOSS
Epoch
0.0
0.2
0.4
0.6
0.8
0
10
20
30
40
50


ACCURACY EVOLUTION - CNN(32,64) - MLP(512,128,10) - Fashion MNIST
train accuracy
test accuracy
Accuracy
0.92
0.94
0.96
0.98
1.00
0
10
20
30
40
50

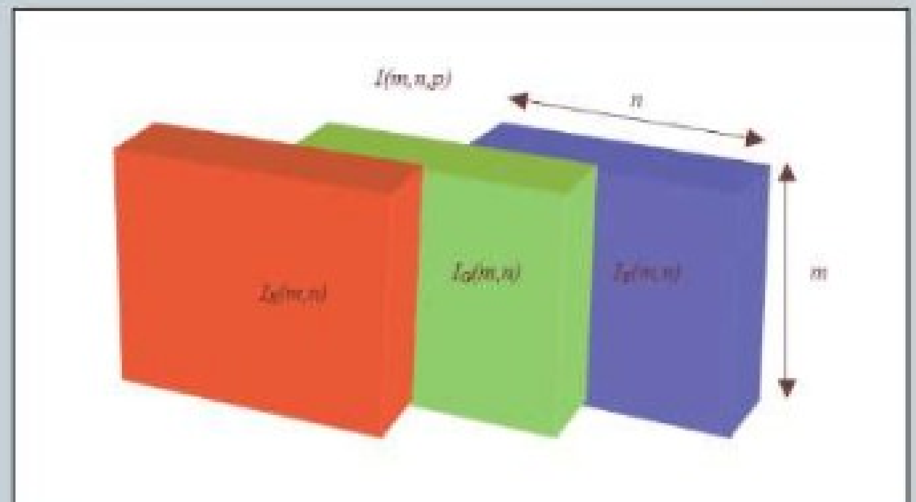
Imágenes a color

$$I_R(m,n,1)=\begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ r_{21} & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{m1} & r_{m2} & \cdots & r_{mn} \end{bmatrix} \quad I_G(m,n,2)=\begin{bmatrix} g_{11} & g_{12} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & g_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{m1} & g_{m2} & \cdots & g_{mn} \end{bmatrix} \quad I_B(m,n,3)=\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$

1=R

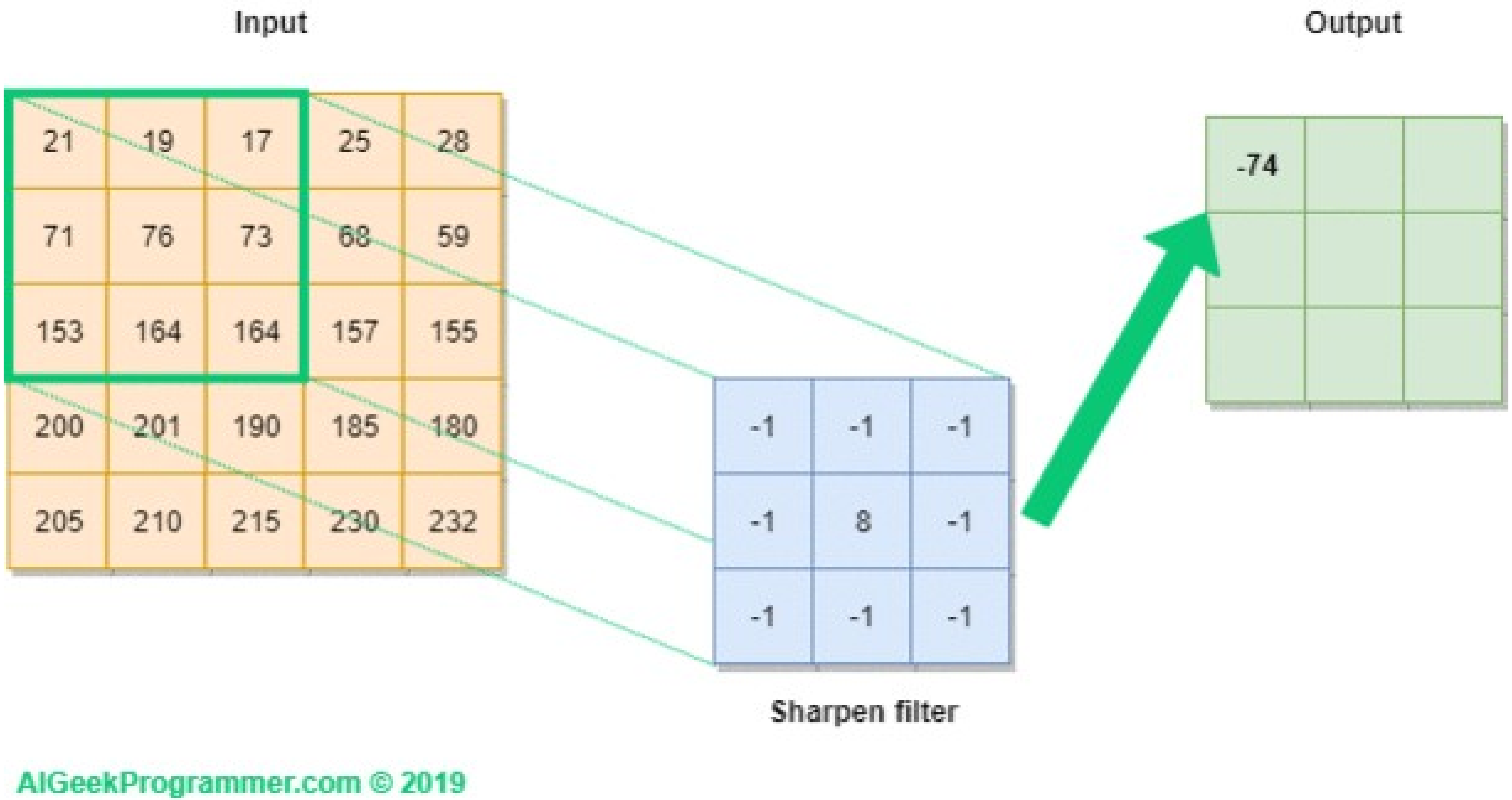
2=G

3=B



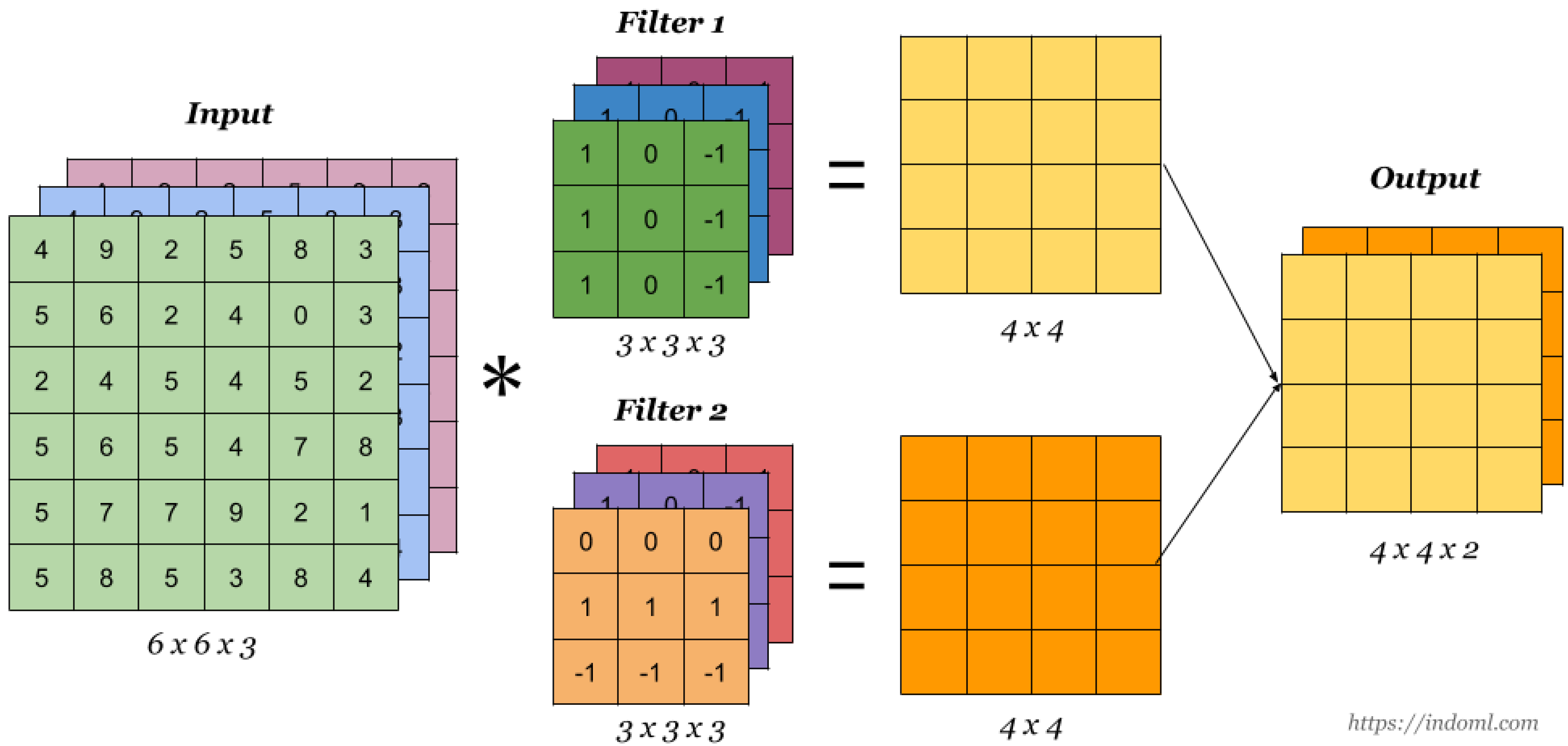
- Vector tensor 2D: (samples, features)
- Series temporales tensor 3D: (samples, timesteps, features)
- Imágenes tensors 4D, ejemplos:
  (sample, height, width, channel) or (sample, channel, height, width)
- Vídeo tensor 5D:
  (sample, frame, height, width, channel) o (sample, frame, channel, height, width)

Convolución 2D:



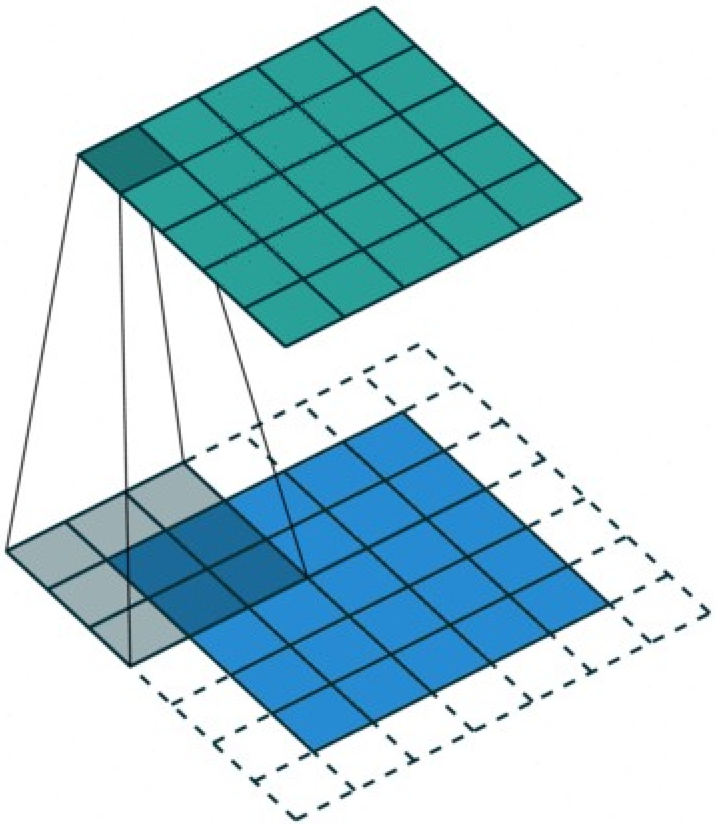
Se aprovecha la información 2D

Convolución 2D (RGB)

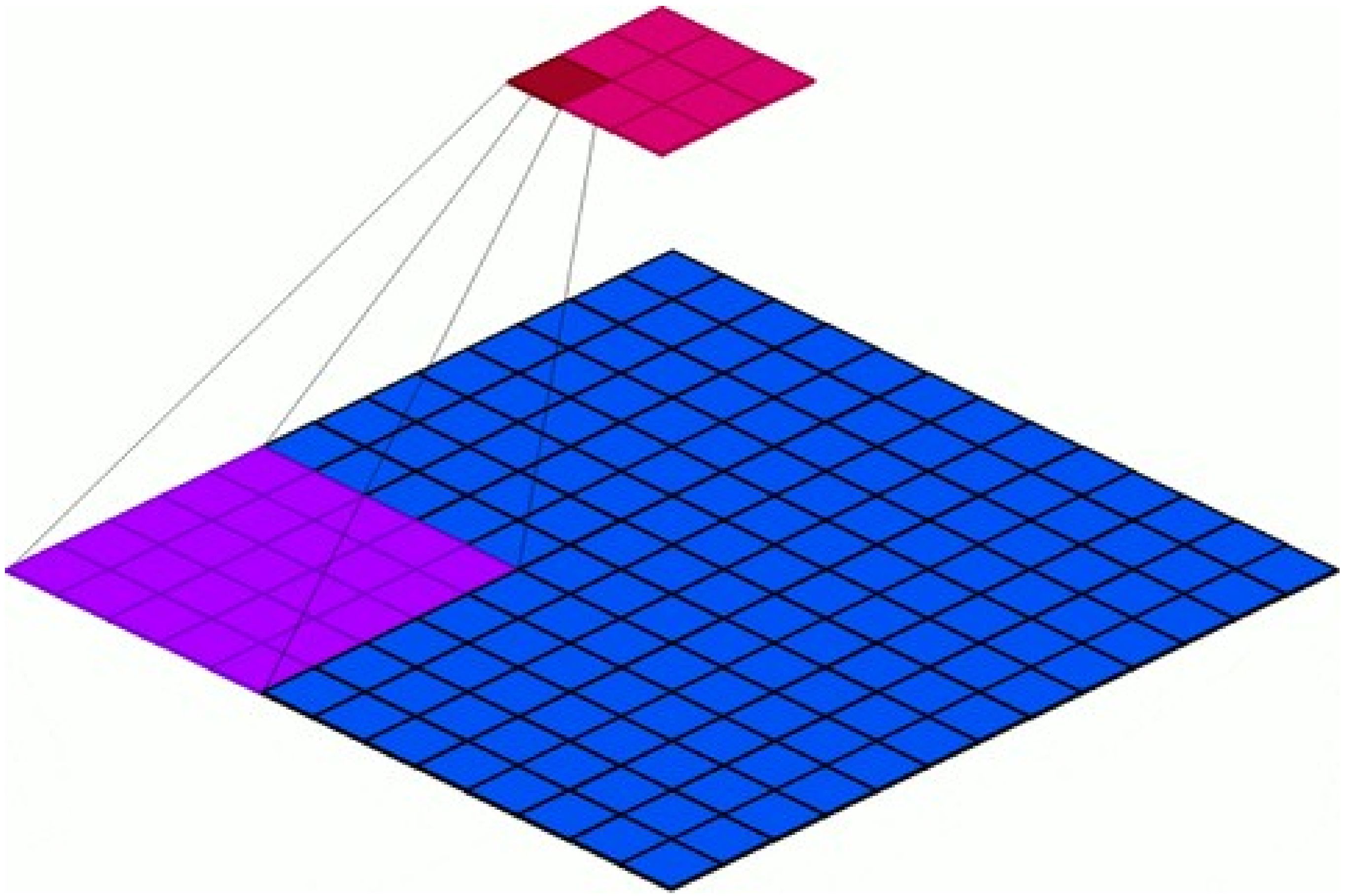


Capa convolucional – entrada: 3 canales – salida: 2 canales

Avance o “Stride”

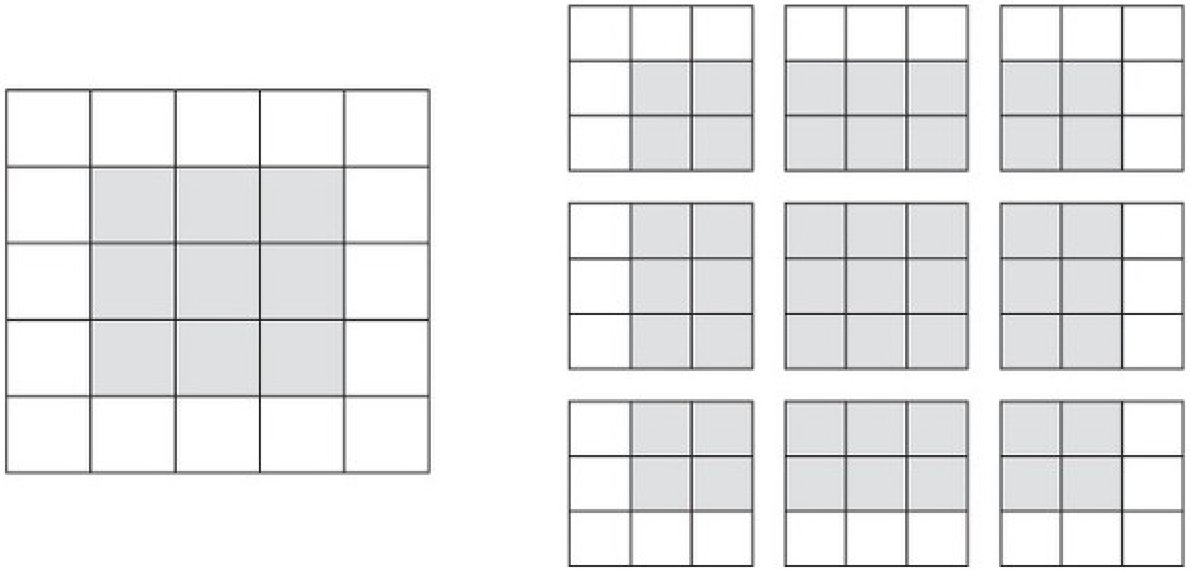


- Núcleo 3x3
- Imagen original arriba
- Padding
- Stride = 1



- Núcleo 5x5
- Imagen original abajo
- No Padding
- Stride = 4

Avance del recorrido del núcleo

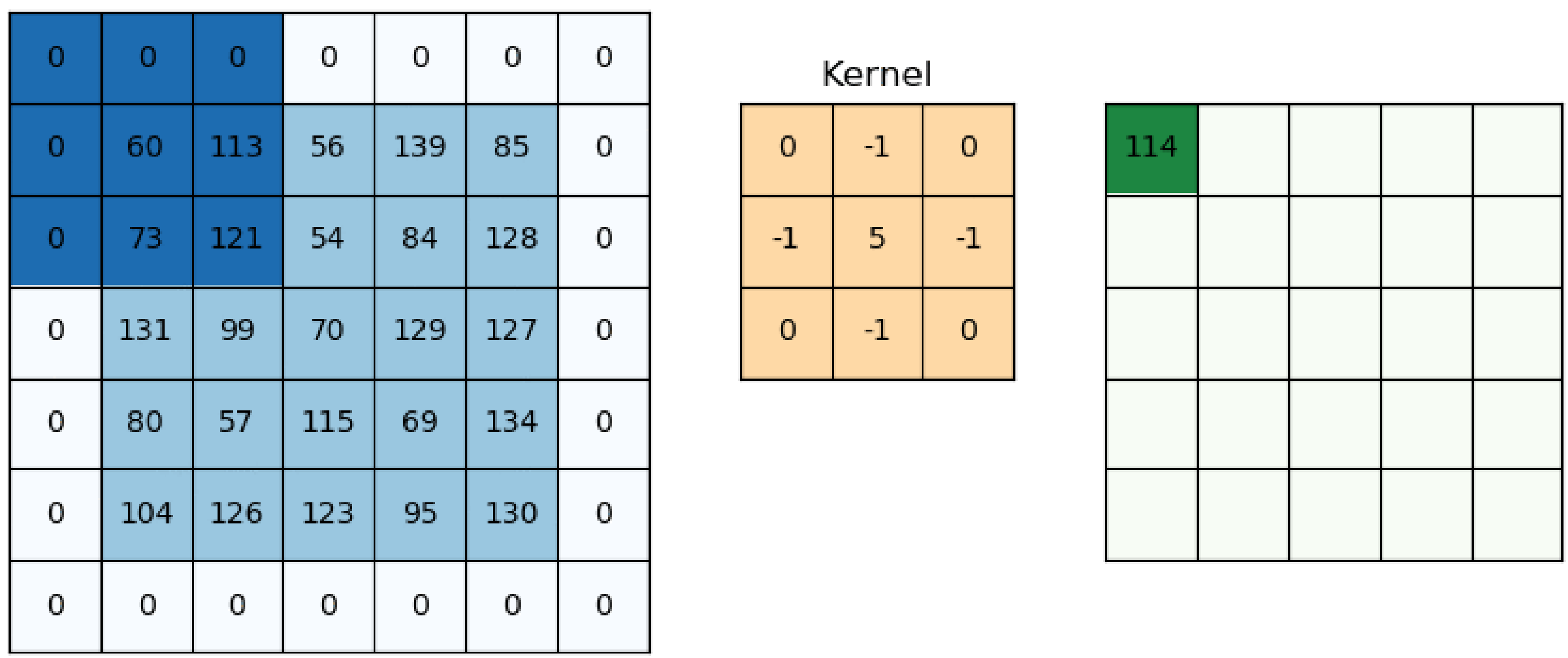


- Tamaño: 3x3, 5x5, 7x7, 9x9, etc.
- Avance (**stride**):
  - De uno en uno (o más)
  - Hasta del tamaño del núcleo
- Se recorre izquierda a derecha
- De arriba a abajo

- Avance horizontal/vertical indep.

- Al llegar a los extremos:
  - Si no hay pixels suficientes
  - Se rellenan artificialmente (**padding**) con pixeles transparentes.

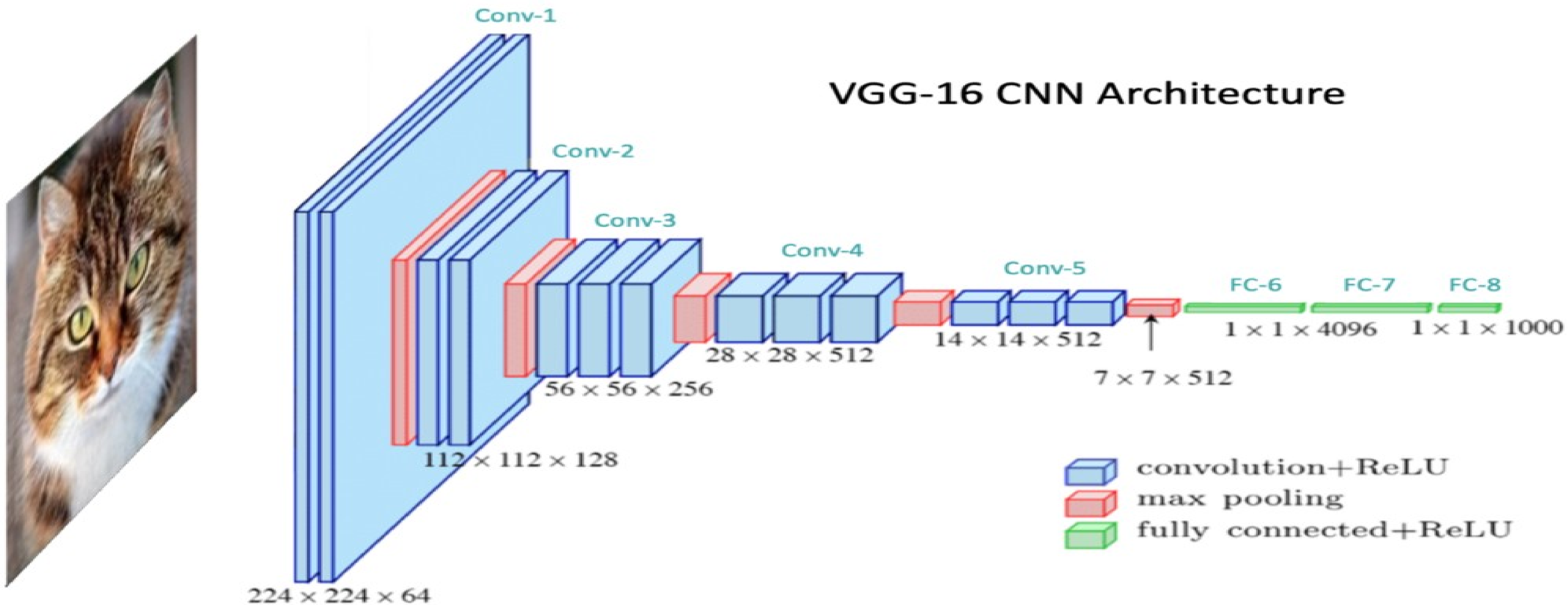
Relleno o “Padding”

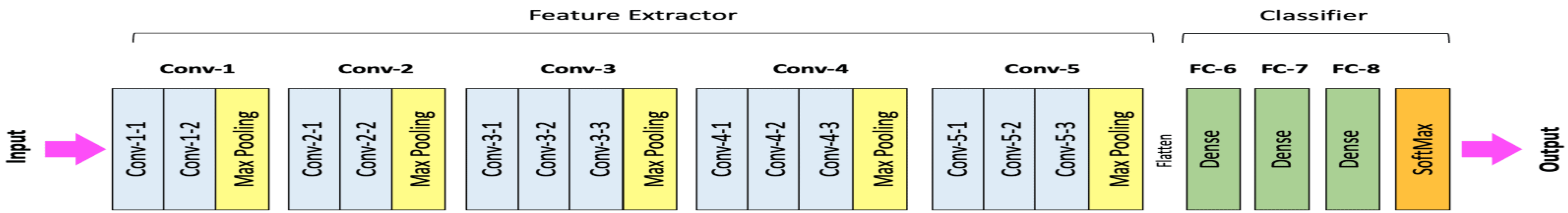


Relleno más habitual:un borde de tamaño ajustado de ceros ajustado al avance

Rellenos más complejos: basados en interpolaciones

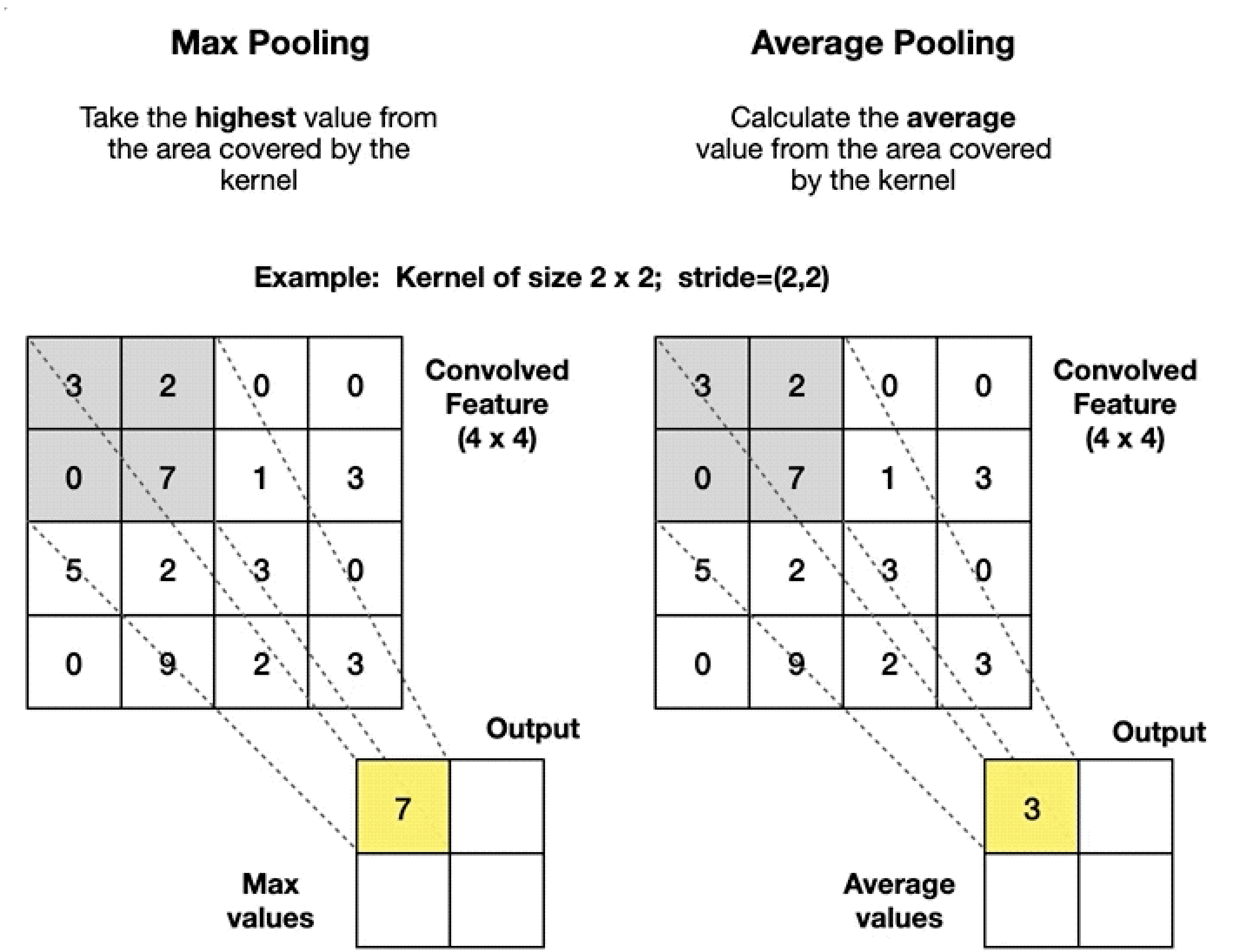
Ejemplo CNN: VGG-16 (channel->last)





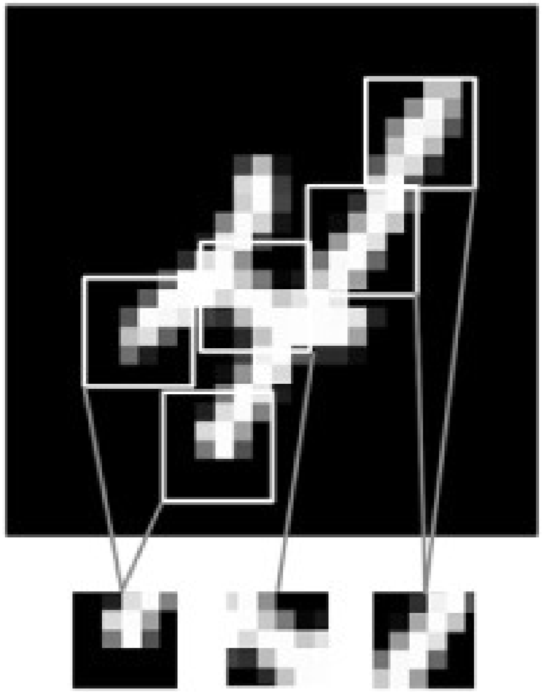
https://www.kaggle.com/code/datastrophy/vgg16-pytorch-implementation

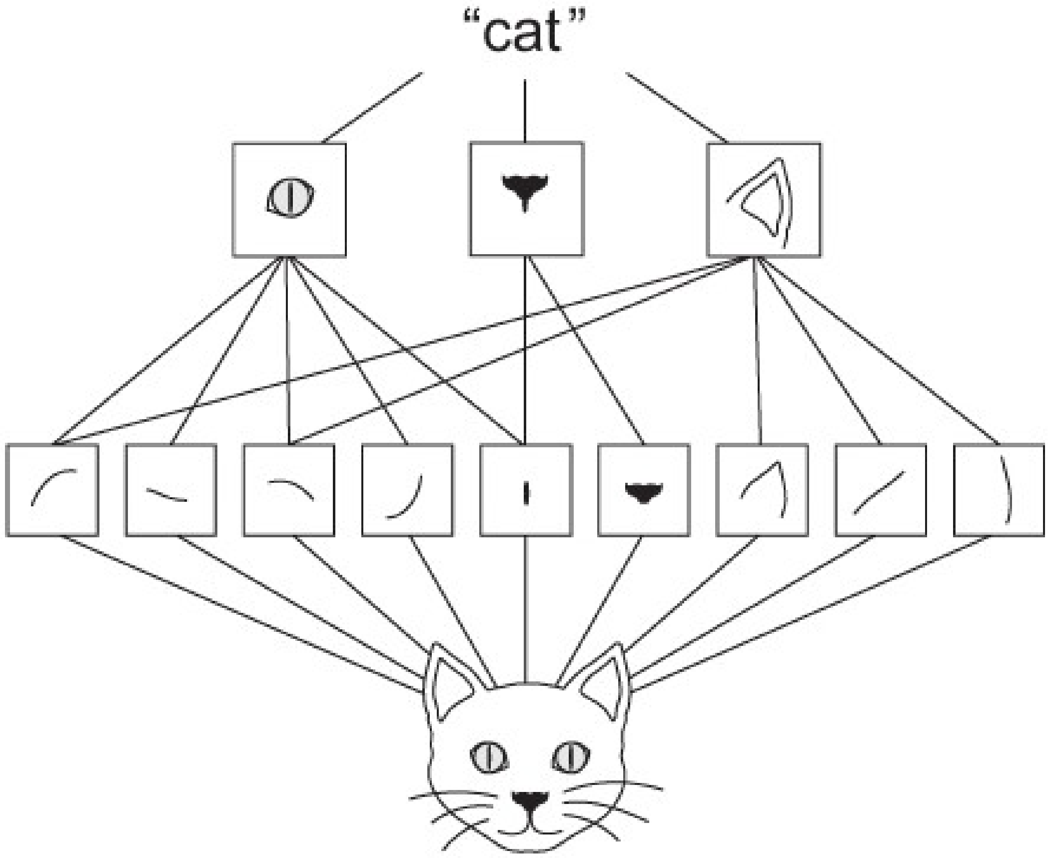
Pooling



- MaxPooling es el más usado
- El tamaño del núcleo habitual es 2x2
  - Podría ser 3x4, por ejemplo, pero no aparece en las implementaciones
- El avance normal igual a tamaño del núcleo (NxN)
  - Nunca supera a N
  - El más pequeño es uno

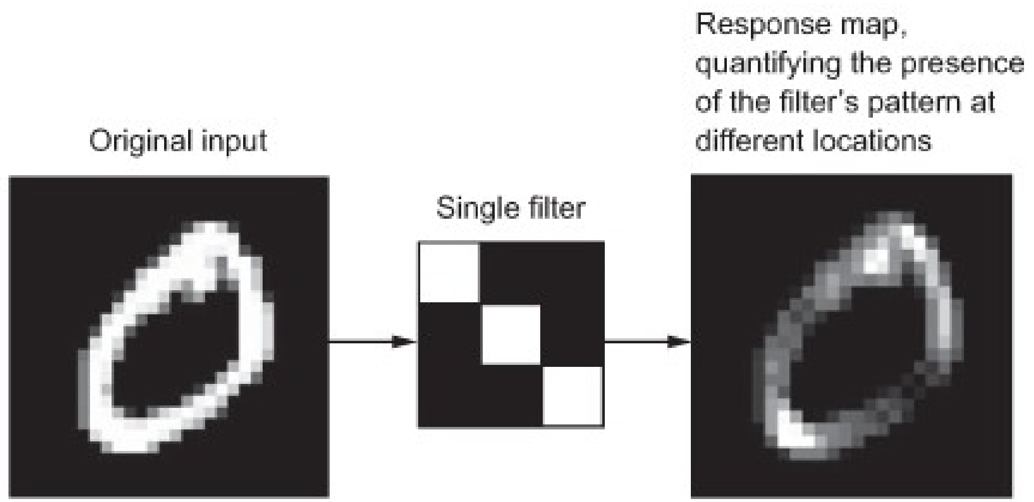
Motivación de la Convolución 2D

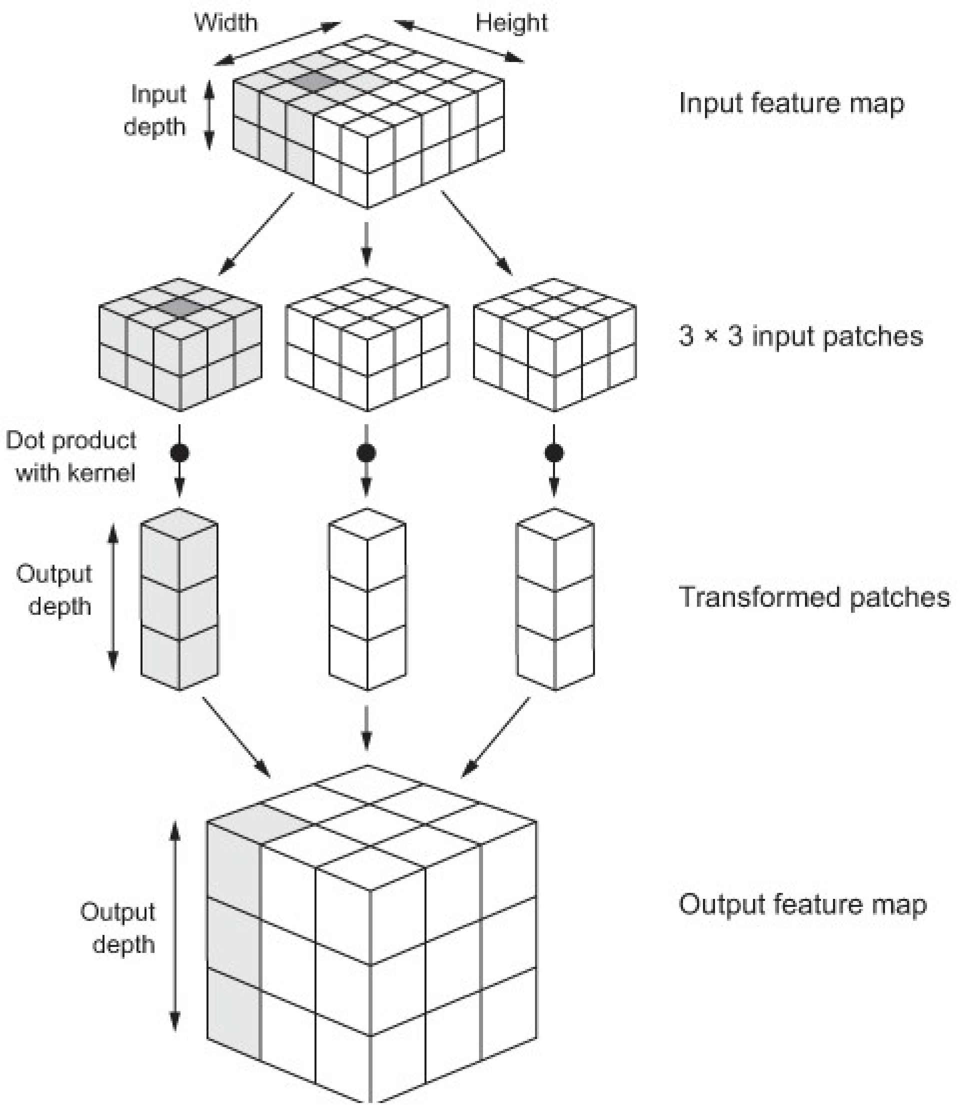




- Reconocimiento de formas simples en las capas más profundas
- Composición de éstas en la sucesivas capas hasta formar patrones (reconocer) más complejos

Tratamiento matricial de la Convolución





Datasets de Pytorch y Dataloaders

```
train_set = torchvision.datasets.FashionMNIST(root = './data/FashionMNIST', download = True, train = True,
                                              transform = transforms.Compose([transforms.ToTensor()]))

test_set = torchvision.datasets.FashionMNIST(root = './data/FashionMNIST', download=True, train=False,
                                             transform = transforms.Compose([transforms.ToTensor()]))
```

https://pytorch.org/vision/0.9/transforms.html

```
train_loader = torch.utils.data.DataLoader(train_set,shuffle=True, batch_size=bandeja)
train_accuracy_loader = torch.utils.data.DataLoader(train_set, batch_size=60000)
test_loader = torch.utils.data.DataLoader(test_set,batch_size=10000)
```

Datasets y Dataloaders en pytorch

Origen de los datos:
- Dataframes
- Matrices (numpy)
- Tensores (torch)
- Ficheros

→

Dataset:
- Empareja entrada/salida
- Operaciones

→

Dataloaders:
Suministro: Memoria o disco
Organiza en bandejas

Datasets creados a partir de tensores

```
class Flatten_Dataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels
    def __len__(self):
        return self.images.shape[0]
    def __getitem__(self, idx):
        return self.images[idx], self.labels[idx]
```

```
train_images = torch.zeros((len(train_image_set), 28*28), dtype=torch.float32)
train_labels = torch.empty(len(train_image_set), dtype=torch.long)

for i, muestra in enumerate(train_image_set):
    train_images[i] = torch.flatten(muestra[0])
    train_labels[i] = muestra[1]
```

Creación de la Red Neuronal

```
class Network(nn.Module):
    def __init__(self):
        super(Network,self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1,out_channels=16,kernel_size=3)
        self.conv2 = nn.Conv2d(in_channels=16,out_channels=32,kernel_size=3)

        self.conv3 = nn.Conv2d(in_channels=32,out_channels=64,kernel_size=3)
        self.conv4 = nn.Conv2d(in_channels=64,out_channels=128,kernel_size=3)

        self.relu = nn.LeakyReLU()

        self.pool = nn.MaxPool2d(kernel_size=2,stride=2)

        self.fc1 = nn.Linear(in_features=4*4*128,out_features=128)
        self.out = nn.Linear(in_features=128,out_features=10)

    def forward(self,x):
        #first CNN layer
        x = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool(x)

        #second CNN layer
        x = self.conv3(x)
        x = self.relu(x)
        x = self.conv4(x)
        x = self.relu(x)
        x = self.pool(x)

        #mlp hidden layer
        x = torch.flatten(x, start_dim=1)
        x = self.fc1(x)
        x = self.relu(x)

        #output layer
        x = self.out(x)
        return x
```

```
xx, yy = next(iter(train_loader))

x1 = model.pool(model.relu(model.conv2(model.relu(model.conv1(xx)))))
x2 = model.pool(model.relu(model.conv4(model.relu(model.conv3(x1)))))
x2.shape
```

```
torch.Size([32, 128, 4, 4])
```

```
summary(model, input_size= (1,28,28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
         LeakyReLU-1            [-1, 1, 28, 28]               0
         LeakyReLU-2            [-1, 1, 28, 28]               0
            Conv2d-3           [-1, 16, 26, 26]             160
         LeakyReLU-4           [-1, 16, 26, 26]               0
            Conv2d-5           [-1, 32, 24, 24]           4,640
         LeakyReLU-6           [-1, 32, 24, 24]               0
         MaxPool2d-7           [-1, 32, 12, 12]               0
            Conv2d-8           [-1, 64, 10, 10]          18,496
         LeakyReLU-9           [-1, 64, 10, 10]               0
           Conv2d-10            [-1, 128, 8, 8]          73,856
        LeakyReLU-11            [-1, 128, 8, 8]               0
        MaxPool2d-12            [-1, 128, 4, 4]               0
           Linear-13                  [-1, 128]         262,272
        LeakyReLU-14                  [-1, 128]               0
        LeakyReLU-15                  [-1, 128]               0
           Linear-16                   [-1, 10]           1,290
================================================================
Total params: 360,714
Trainable params: 360,714
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.73
Params size (MB): 1.38
Estimated Total Size (MB): 2.11
----------------------------------------------------------------
```

Aprendizaje por épocas

```
X_train, y_train = next(iter(train_accuracy_loader))
X_test, y_test = next(iter(test_loader))
```

```
for i in range(epochs):
    print("Epoch: ", i)
    model.train(True)
    for images,targets in tqdm(train_loader):
        #making predictions
        y_pred = model(images)

        #calculating loss
        loss = criterion(y_pred,targets.long())

        #backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


    with torch.no_grad():
        y_train_pred = model(X_train)
        train_accuracy.append(np.mean((y_train == y_train_pred.argmax(dim=1)).numpy()))
        train_loss.append(criterion(y_train_pred, y_train.long()).numpy())
    print('Train Loss: {0:.5f}'.format(train_loss[-1]), '\tTrain Accuracy: {0:.5f}'.format(train_accuracy[-1]))

    with torch.no_grad():
        y_test_pred = (model(X_test))
        test_accuracy.append(np.mean((y_test == y_test_pred.argmax(dim=1)).numpy()))
        test_loss.append(criterion(y_test_pred, y_test.long()).numpy())

    learning_rate.append(optimizer.param_groups[0]['lr'])
    print('  Val Loss: {0:.5f}'.format(test_loss[-1]), '\t Test Accuracy: {0:.5f}'.format(test_accuracy[-1]), "\tLearning Rate:", learning_rate[-1])
```

Falta de RAM tasa aciertos

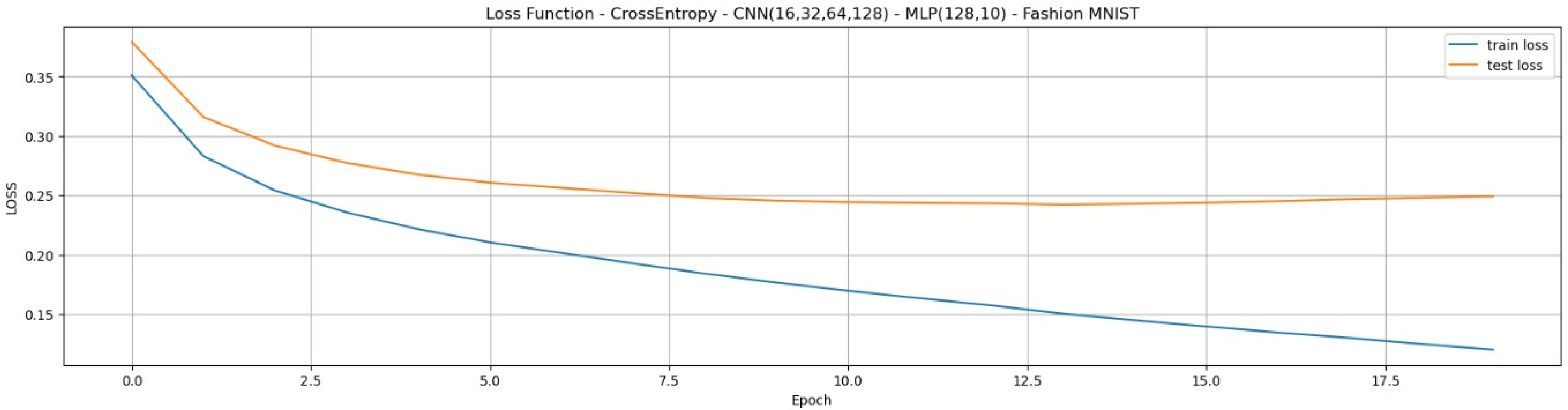
```
with torch.no_grad():
        aciertos = 0
        muestras = 0
        perdidas = 0.0
        for X_train, y_train in train_accuracy_loader:
            y_train_pred = model(X_train)
            aciertos += np.sum((y_train == y_train_pred.argmax(dim=1)).numpy()))
            muestras += X_train.shape[0]
            perdidas += criterion(y_train_pred, y_train.long()).numpy()

train_accuracy.append(aciertos/muestras)
train_loss.append(perdidas)

print('Train Loss: {0:.5f}'.format(train_loss[-1]), '\tTrain Accuracy: {0:.5f}'.format(train_accuracy[-1]))

with torch.no_grad():
        aciertos = 0
        muestras = 0
        perdidas = 0.0
        for X_test, y_test in test_loader:
            y_test_pred = (model(X_test))
            aciertos += np.sum((y_test == y_test_pred.argmax(dim=1)).numpy()))
            muestras += X_test.shape[0]
            perdidas += criterion(y_test_pred, y_test.long()).numpy()

test_accuracy.append(aciertos/muestras)
test_loss.append(perdidas)


learning_rate.append(optimizer.param_groups[0]['lr'])
print('  Val Loss: {0:.5f}'.format(test_loss[-1]), '\t Test Accuracy: {0:.5f}'.format(test_accuracy[-1]), "\tLearning Rate:", learning_rate[-1])
```

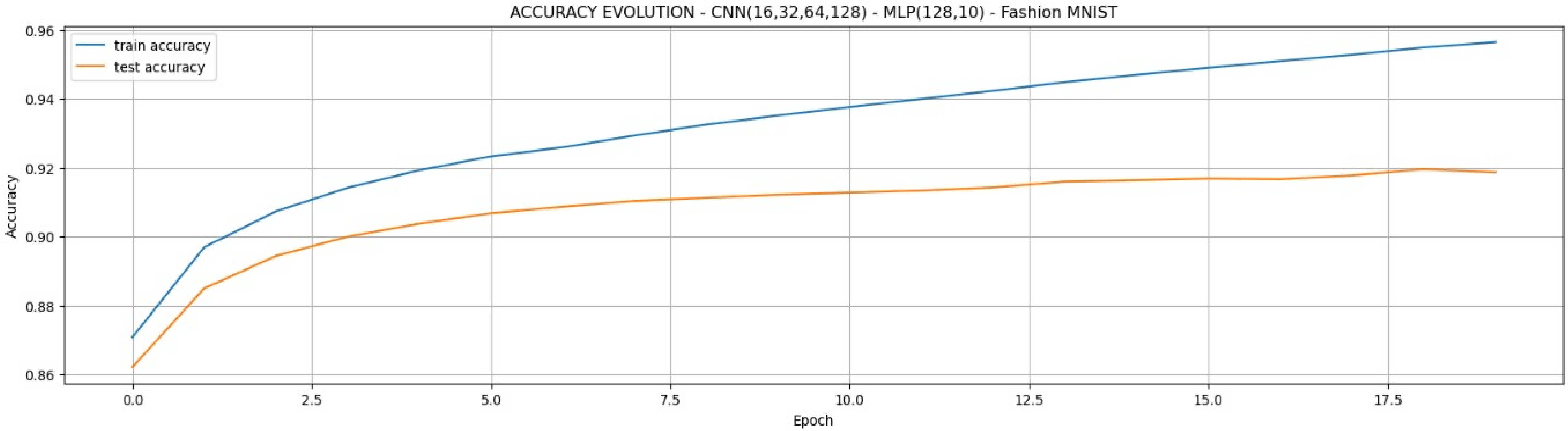
```
train_loader = torch.utils.data.DataLoader(train_set,shuffle=True, batch_size=bandeja)
train_accuracy_loader = torch.utils.data.DataLoader(train_set, batch_size=lote_train)
test_loader = torch.utils.data.DataLoader(test_set,batch_size=lote_test)
```

Ejercicio (entrega):

- Clasificación de imágenes:
  - Dataset: MNIST FASHION incluida en PyTorch
  - Arquitectura VGG:
    - CNN(1,16) + CNN(16,32) + MaxPool(2)
    - CNN(32,64) + CNN(64,128) + MaxPool(2)
    - MLP(¿?, 128, 10)
  - Tamaño de bandeja: 32
  - Función de pérdida: Entropía Cruzada
  - Optimizador: Adagrad (tasa de aprendizaje adaptativo)
  - Épocas:20

Resultados Práctica: Fashion-MNIST





VGG Pre-entrenado

- Las imágenes tienen que ser:

  224 x 224 x 3 canales (channel last)

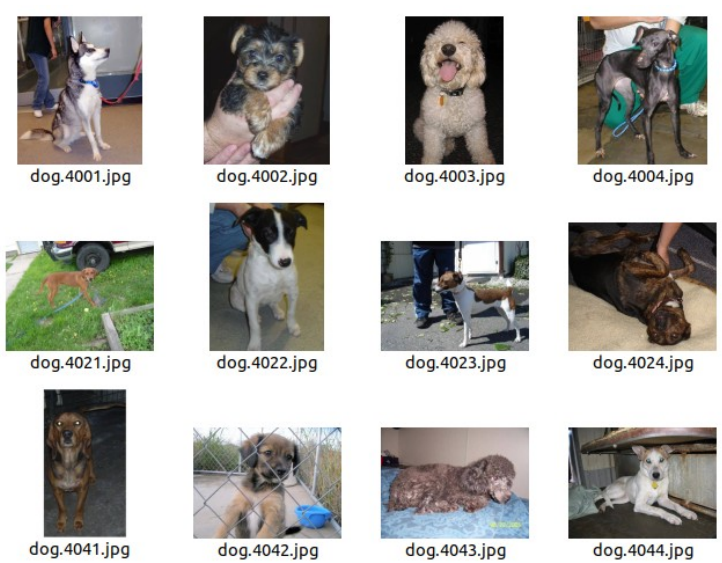
- Deben escalarse para que:
  - Media por canal: 0.485, 0.456, 0.406
  - Desviación estándar por canal: 0.229, 0.224, 0.225

- El modelo tiene 14 ~ 15 millones de pesos.

- De todo esto se concluye que el dataset de imágenes no se va a poder suministrar en matrices, como MNIST, CIFAR, etc.

Conjunto de Imágenes en ficheros

- Se organizan en carpetas: “training”, “test” y, a veces, “validate”.
- Dentro de estos **directorios** aparecen otros tantos correspondiente a las **CLASES**
- Ejemplo de clasificación binaria: perros y gatos



Dataset de ficheros

```python
train_data_dir = './data/cats-and-dogs/training_set/training_set/'
test_data_dir = './data/cats-and-dogs/test_set/test_set/'
```

```python
class CatsDogs(Dataset):
    def __init__(self, folder):
        cats = glob(folder+'/cats/*.jpg')
        dogs = glob(folder+'/dogs/*.jpg')
        self.fpaths = cats[:500] + dogs[:500]
        self.normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])
        from random import shuffle, seed; seed(10);
        shuffle(self.fpaths)
        self.targets =[fpath.split('/')[-1].startswith('dog') for fpath in self.fpaths]

    def __len__(self): return len(self.fpaths)

    def __getitem__(self, ix):
        f = self.fpaths[ix]
        target = self.targets[ix]
        im = (cv2.imread(f)[:,:,::-1])
        im = cv2.resize(im, (224,224))
        im = torch.tensor(im/255)
        im = im.permute(2,0,1)
        im = self.normalize(im)
        return im.float().to(device), torch.tensor([target]).float().to(device)
```

```python
data = CatsDogs(train_data_dir)
```

Manejo de las partes VGG16

- Fijar la extracción de características: CNN

- Adaptar el clasificador: MLP

```
model = models.vgg16(weights='DEFAULT')
for param in model.parameters():
    param.requires_grad = False
model.avgpool = nn.AdaptiveAvgPool2d(output_size=(1,1))
model.classifier = nn.Sequential(nn.Flatten(),
                                 nn.Linear(512, 128),
                                 nn.ReLU(),
                                 nn.Dropout(0.2),
                                 nn.Linear(128, 1),
                                 nn.Sigmoid())
loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(),lr= 1e-3)
```

Parámetros de VGG16

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 512, 7, 7]           --
|    └─Conv2d: 2-1                       [-1, 64, 224, 224]        (1,792)
|    └─ReLU: 2-2                         [-1, 64, 224, 224]        --
|    └─Conv2d: 2-3                       [-1, 64, 224, 224]        (36,928)
|    └─ReLU: 2-4                         [-1, 64, 224, 224]        --
|    └─MaxPool2d: 2-5                    [-1, 64, 112, 112]        --
|    └─Conv2d: 2-6                       [-1, 128, 112, 112]       (73,856)
|    └─ReLU: 2-7                         [-1, 128, 112, 112]       --
|    └─Conv2d: 2-8                       [-1, 128, 112, 112]       (147,584)
|    └─ReLU: 2-9                         [-1, 128, 112, 112]       --
|    └─MaxPool2d: 2-10                   [-1, 128, 56, 56]         --
|    └─Conv2d: 2-11                      [-1, 256, 56, 56]         (295,168)
|    └─ReLU: 2-12                        [-1, 256, 56, 56]         --
|    └─Conv2d: 2-13                      [-1, 256, 56, 56]         (590,080)
|    └─ReLU: 2-14                        [-1, 256, 56, 56]         --
|    └─Conv2d: 2-15                      [-1, 256, 56, 56]         (590,080)
|    └─ReLU: 2-16                        [-1, 256, 56, 56]         --
|    └─MaxPool2d: 2-17                   [-1, 256, 28, 28]         --
|    └─Conv2d: 2-18                      [-1, 512, 28, 28]         (1,180,160)
|    └─ReLU: 2-19                        [-1, 512, 28, 28]         --
|    └─Conv2d: 2-20                      [-1, 512, 28, 28]         (2,359,808)
|    └─ReLU: 2-21                        [-1, 512, 28, 28]         --
|    └─Conv2d: 2-22                      [-1, 512, 28, 28]         (2,359,808)
|    └─ReLU: 2-23                        [-1, 512, 28, 28]         --
|    └─MaxPool2d: 2-24                   [-1, 512, 14, 14]         --
|    └─Conv2d: 2-25                      [-1, 512, 14, 14]         (2,359,808)
|    └─ReLU: 2-26                        [-1, 512, 14, 14]         --
|    └─Conv2d: 2-27                      [-1, 512, 14, 14]         (2,359,808)
|    └─ReLU: 2-28                        [-1, 512, 14, 14]         --
|    └─Conv2d: 2-29                      [-1, 512, 14, 14]         (2,359,808)
|    └─ReLU: 2-30                        [-1, 512, 14, 14]         --
|    └─MaxPool2d: 2-31                   [-1, 512, 7, 7]           --
├─AdaptiveAvgPool2d: 1-2                 [-1, 512, 1, 1]           --
├─Sequential: 1-3                        [-1, 1]                   --
|    └─Flatten: 2-32                     [-1, 512]                 --
|    └─Linear: 2-33                      [-1, 128]                 65,664
|    └─ReLU: 2-34                        [-1, 128]                 --
|    └─Dropout: 2-35                     [-1, 128]                 --
|    └─Linear: 2-36                      [-1, 1]                   129
|    └─Sigmoid: 2-37                     [-1, 1]                   --
==========================================================================================
Total params: 14,780,481
Trainable params: 65,793
Non-trainable params: 14,714,688
Total mult-adds (G): 15.36
==========================================================================================
Input size (MB): 0.57
Forward/backward pass size (MB): 103.36
Params size (MB): 56.38
Estimated Total Size (MB): 160.32
==========================================================================================
```

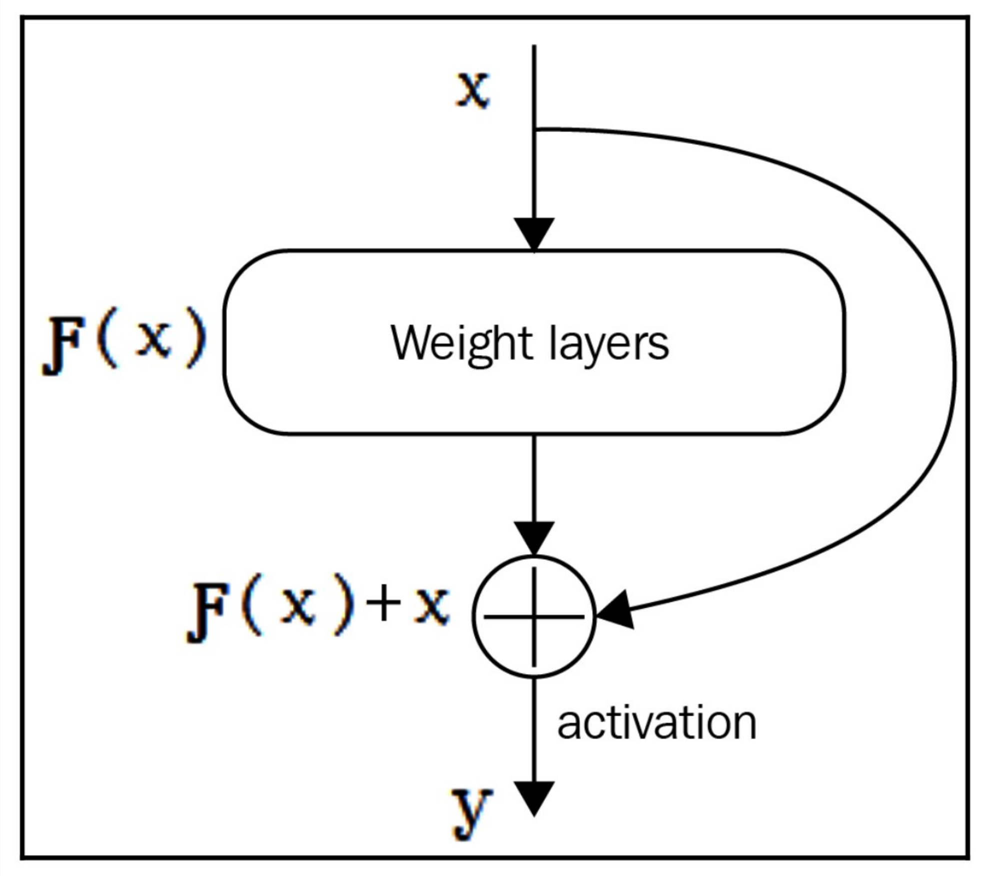

```
Total params: 14,780,481
Trainable params: 65,793
Non-trainable params: 14,714,688
Total mult-adds (G): 15.36
```

Motivación de la red ResNet

- En la fase hacia adelante en Deep Learning:
  - En las últimas capas se **pierde** casi toda la **información** original
- En la fase de retroprogación (aprendizaje):
  - En las primeras capas, el efecto de la **evanescencia** del **gradiente** es muy acusado

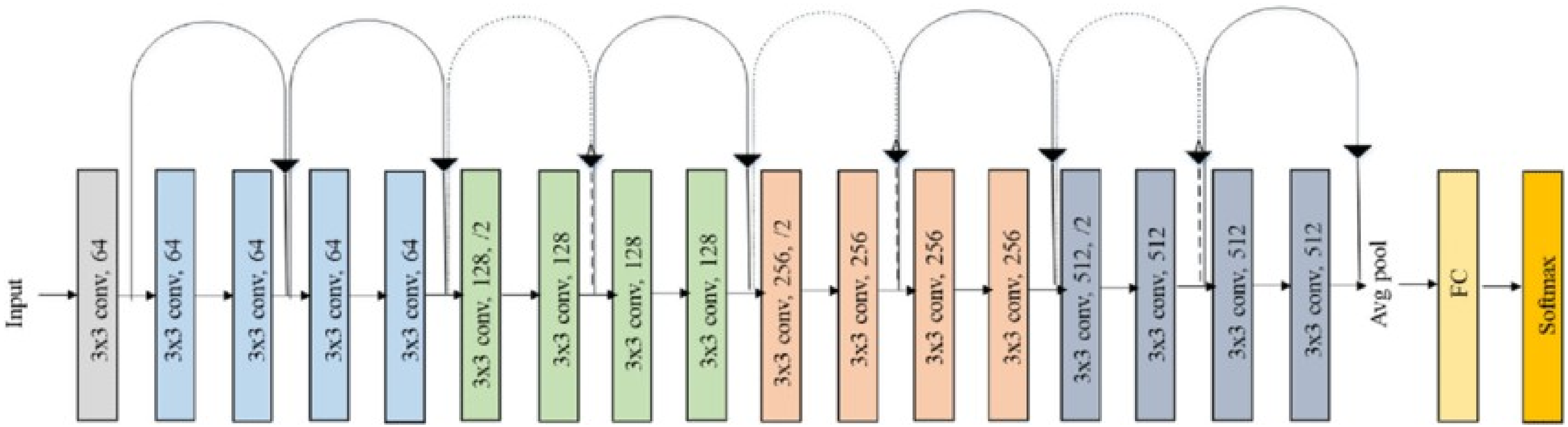
- Solución: añadir una conexión directa, que puentee la entrada con la salida de cada capa (**CONEXIÓN RESIDUAL**)

Conexión Residual




```
class ResLayer(nn.Module):

    def __init__(self,ni,no,kernel_size,stride=1):
        super(ResLayer, self).__init__()
        padding = kernel_size - 2
        self.conv = nn.Sequential(
            nn.Conv2d(ni, no, kernel_size, stride, padding=padding),
            nn.ReLU())

    def forward(self, x):
        x = self.conv(x) + x
        return x
```

Arquitectura ResNet



Extractor de características

Clasificador

Tipos de VGG y ResNet en Pytorch

- https://pytorch.org/vision/0.8/models.html?highlight=vgg

- Simulación:
  - VGG16
  - VGG19
  - ResNet18