CNN – Convolutional neuronal networks

PhD Candidate: Miguel Ángel Medina Ramírez

Deep Learning on Large Images

64x64x3 (color)
Low resolution

12288 features

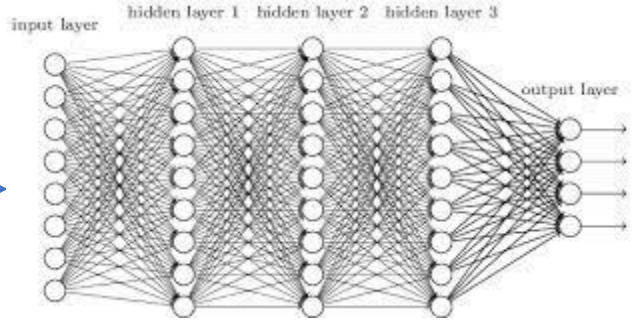
ML Model

Cat? 0/1

¿Cuántos parámetros tendría? => más o menos… 3B para un MLP con buen rendimento.

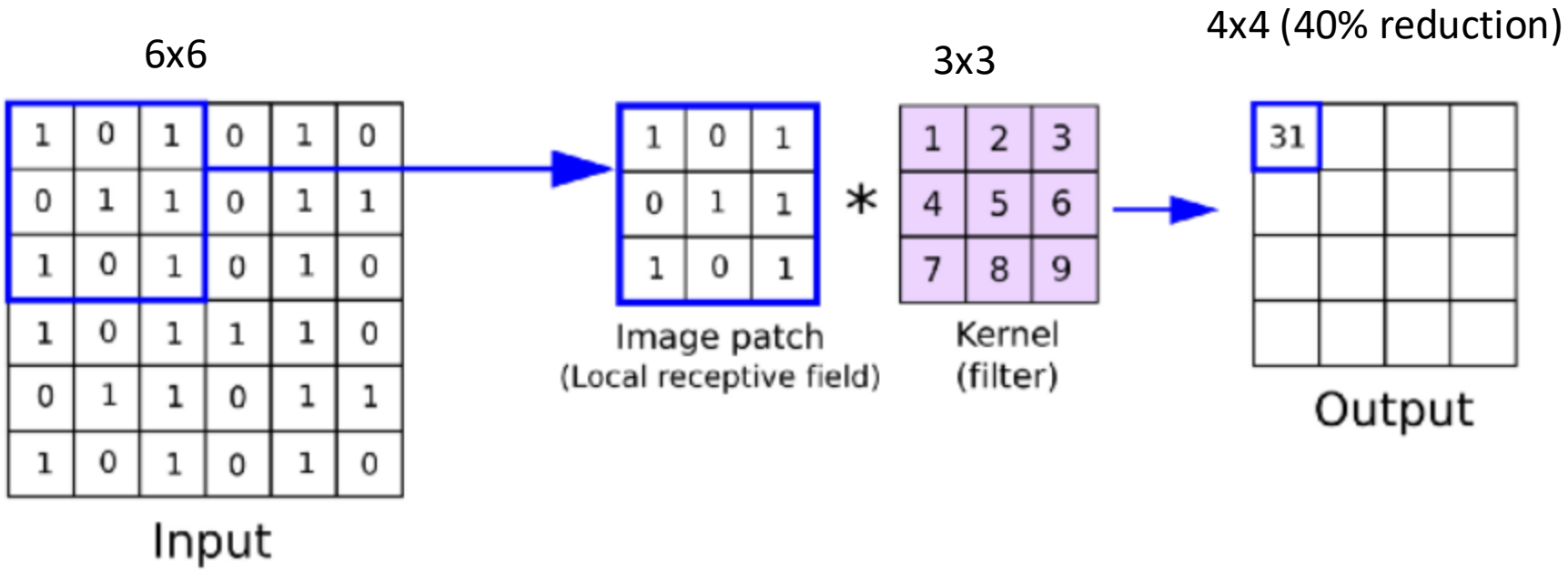
1000x1000x3
High resolution

3M features



Cat? 0/1

Convolution



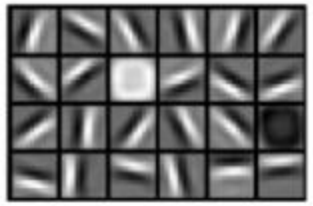
Es un selector de características en imágenes =>

$$conv2D(i,j) = \sum_{y=0}^{2}\sum_{x=0}^{2} kernel(x,y) \cdot imagen(i-x, j-y)$$

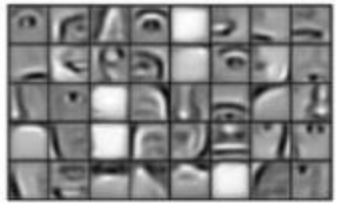
$$conv2D(i,j) = \sum_{y=0}^{2}\sum_{x=0}^{2} kernel(x,y) \cdot imagen(i+x, j+y)$$

Convolution == Filtro de características

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

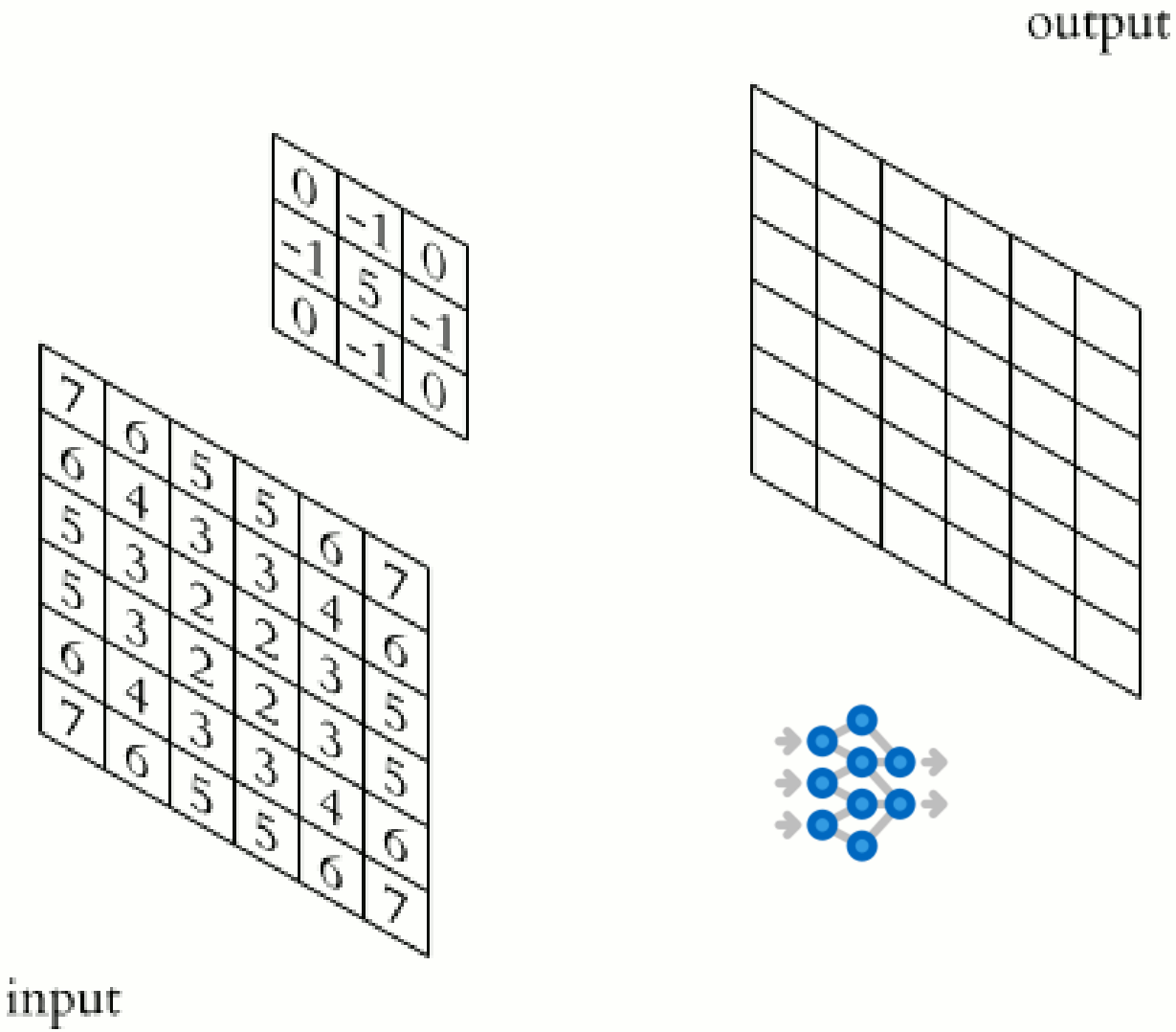


$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$



$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Convolution 2D



Convolution 2D - Pytorch

```python
import torch.nn as nn

conv_layer = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, stride=1, padding=1)
```

- **in_channels=3:** Especifica que la entrada tiene 3 canales, lo que es típico para imágenes en color RGB.
- **out_channels=64:** Especifica que queremos 64 filtros (o kernels) en esta capa, por lo que la salida tendrá 64 canales.
- **kernel_size=3:** Establece el tamaño del filtro a 3x3.
- **padding=1:** Añade un relleno (padding) de 1 píxel alrededor de la imagen, lo que es útil para mantener el tamaño de la imagen después de la convolución, especialmente cuando se usa un kernel de 3x3.
- **stride=1:** Especifica un desplazamiento (stride) de 1 en ambas direcciones (ancho y alto).

Explicación adicional sobre el desplazamiento (stride) y el relleno (padding)

Stride:
El desplazamiento controla cómo se mueve el filtro o kernel a través de la imagen. Un desplazamiento de 1 significa que el filtro se desplaza un píxel a la vez. Si el desplazamiento se establece en 2, entonces el filtro salta 2 píxeles a la vez mientras se desliza. Esto tiene el efecto de reducir las dimensiones espaciales (es decir, ancho y alto) del volumen de salida. Un desplazamiento más grande resulta en una menor dimensión espacial en la salida.

Padding:
El relleno se refiere a agregar píxeles extra alrededor del borde de la imagen. Esto generalmente se hace para controlar las dimensiones espaciales del volumen de salida, especialmente cuando se desea que permanezca igual que el volumen de entrada. Sin relleno, al convolucionar una imagen con un filtro se reducirían sus dimensiones espaciales. Por ejemplo, si tienes una imagen de 5x5 y usas un filtro de 3x3 sin relleno, la salida resultante sería de 3x3. Agregar relleno asegura que el tamaño de salida pueda permanecer igual al tamaño de entrada si así se desea.

¡Con estos dos parámetros podemos garantizar que en vez de filtrar lo que hacemos es **transformar**!

¿Cuándo me interesa reducir o transformar una imagen?

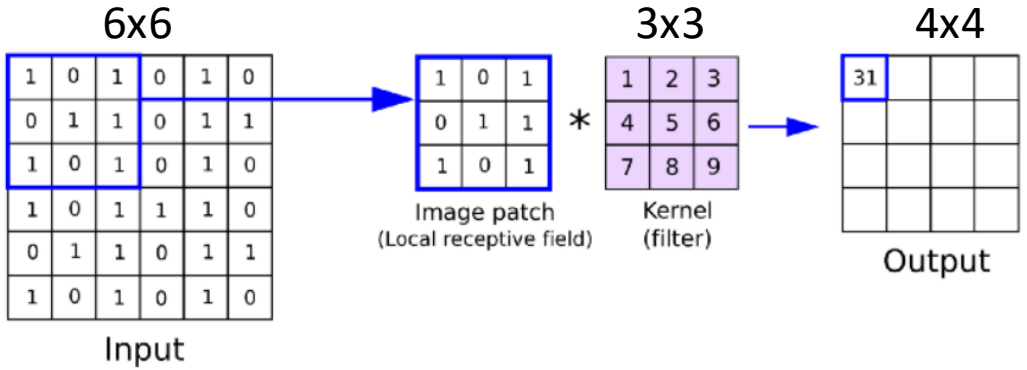
- **Transformar**: Si deseas mantener la resolución espacial a lo largo de múltiples capas, el "same padding" es la opción ideal. Este tipo de padding es útil cuando quieres mantener las dimensiones espaciales a través de varias capas de la red, lo que permite diseñar arquitecturas más profundas sin preocuparse por la disminución rápida del tamaño espacial.
- **Reducir:** Si estás buscando extraer características más abstractas y reducir las dimensiones de tu imagen a medida que se propaga a través de la red (por ejemplo, para llegar a una capa completamente conectada al final), podrías optar por un padding más pequeño o incluso no usar padding.
- En la práctica, muchas arquitecturas populares de CNN utilizan una combinación de ambos enfoques en diferentes etapas de la red.

¿Cómo sabemos las dimensiones del tensor resultante?

$$altura_{resultado} = \frac{(altura_{imagen} + 2p \ - altura_{kernel})}{s} + 1$$

$$ancho_{resultado} = \frac{(ancho_{imagen} + 2p \ - ancho_{kernel})}{s} + 1$$

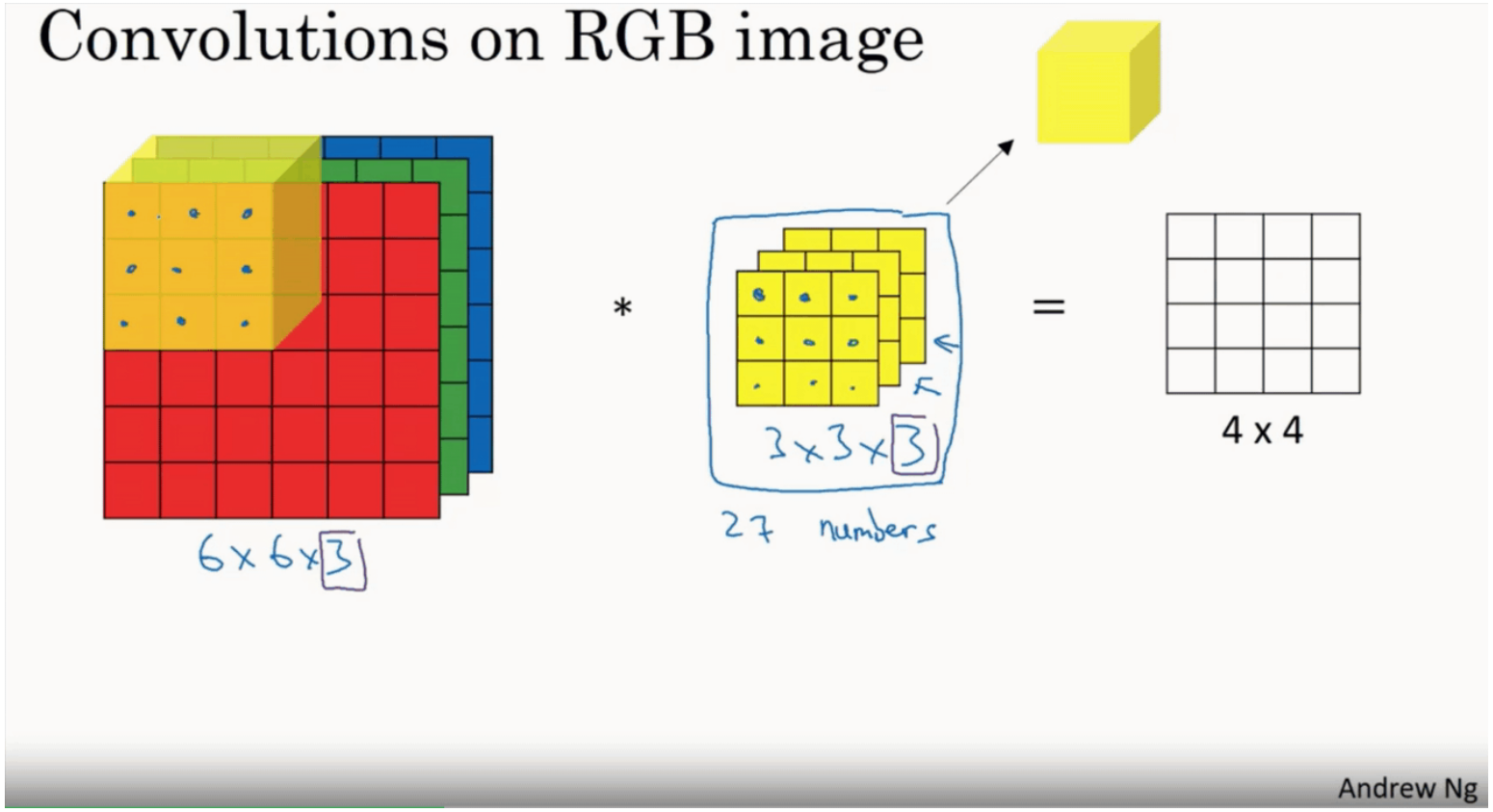
Aquí, el $2p$ es porque el padding se agrega tanto al principio como al final (izquierda y derecha para el ancho, y arriba y abajo para la altura).



$$altura_{resultado} = \frac{(6 + 2 * 0 \ - 3)}{1} + 1 = 3 + 1 = 4$$

$$ancho_{resultado} = \frac{(6 + 2 * 0 \ - 3)}{1} + 1 = 3 + 1 = 4$$

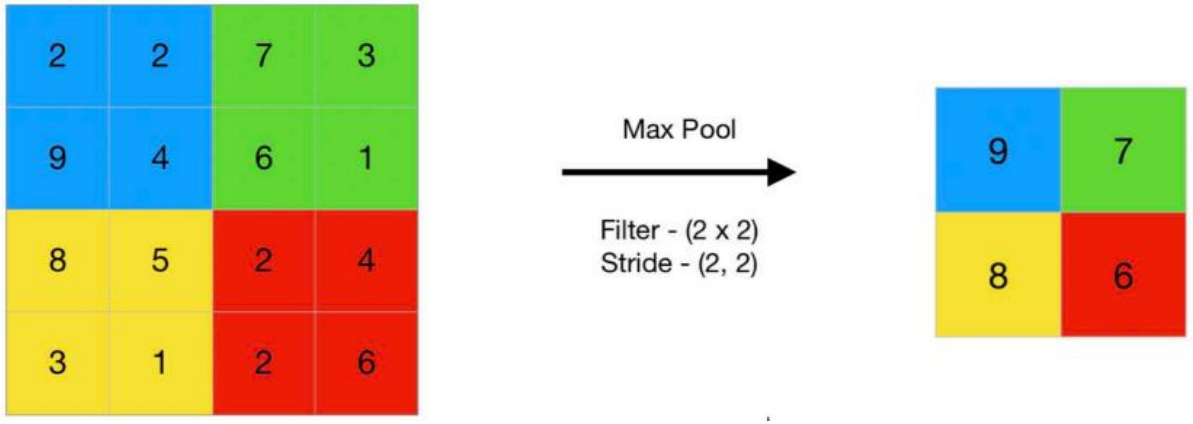
Convolution 2D – Canales?



Pooling

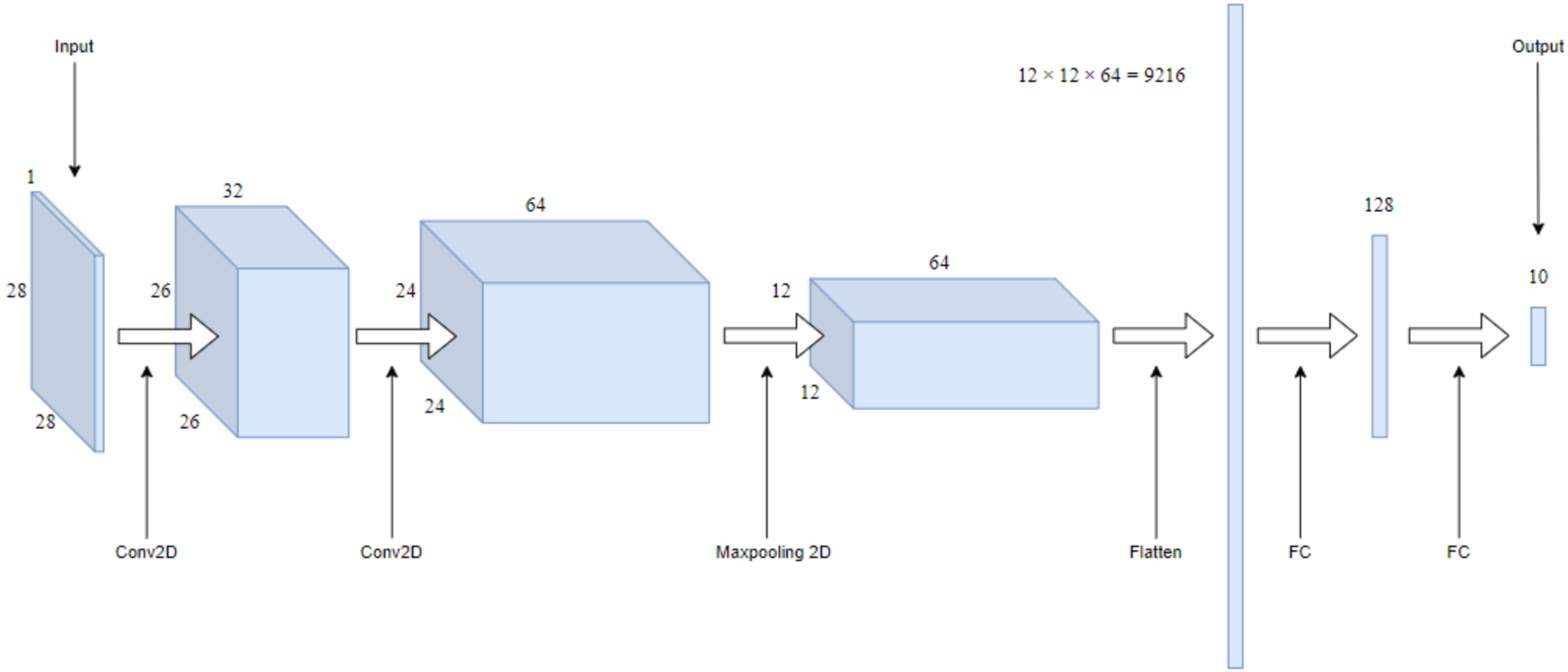
```python
import torch.nn as nn

maxpool_layer = nn.MaxPool2d(kernel_size=2, stride=2)
avgpool_layer = nn.AvgPool2d(kernel_size=2, stride=2)
```

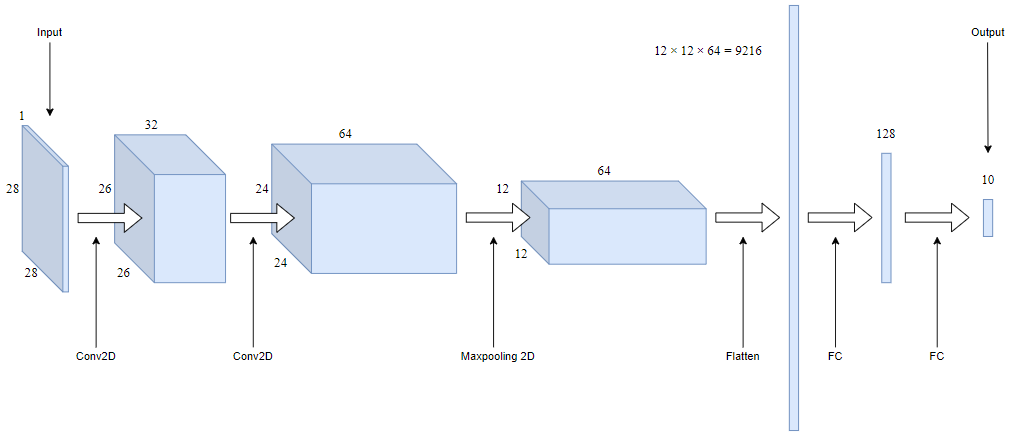


Aviso : Manejar con precaución

Convnet

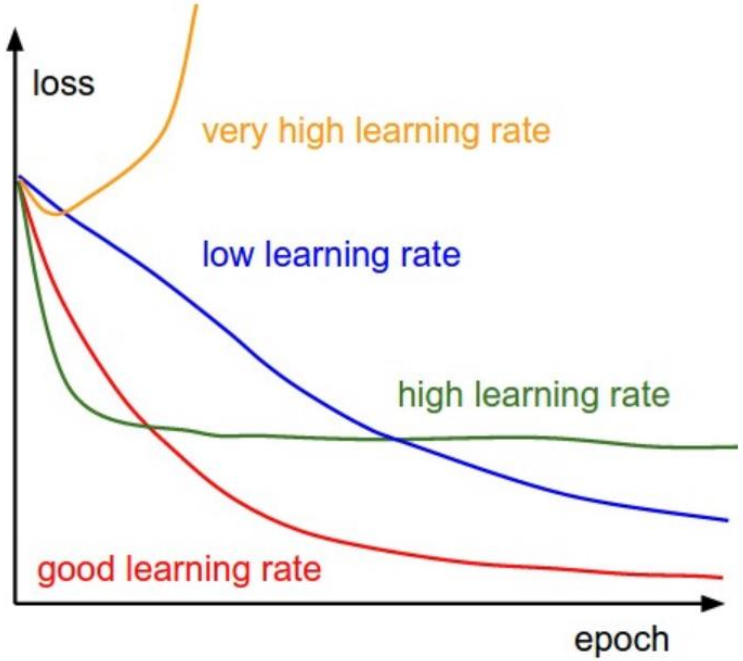

Input
1
28
28
32
26
26
Conv2D
64
24
24
Conv2D
64
12
12
Maxpooling 2D
12 × 12 × 64 = 9216
Flatten
128
FC
Output
10
FC

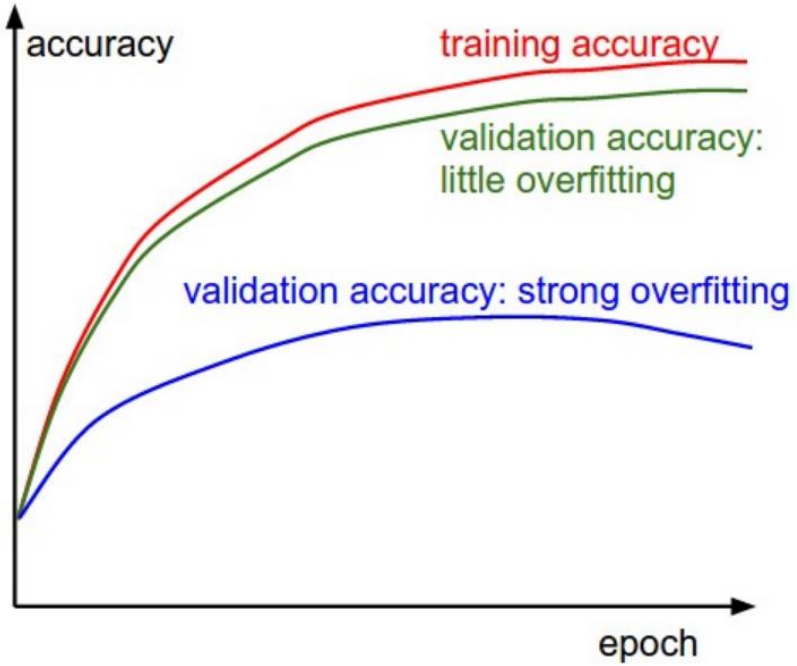
Convnet


Input
1
28
28
32
26
26
Conv2D
64
24
24
Conv2D
12
12
64
Maxpooling 2D
12 × 12 × 64 = 9216
Flatten
128
FC
Output
10
FC


TIPO EXCELSIUS

¿Cómo sé cuándo una red esta bien entrenada?

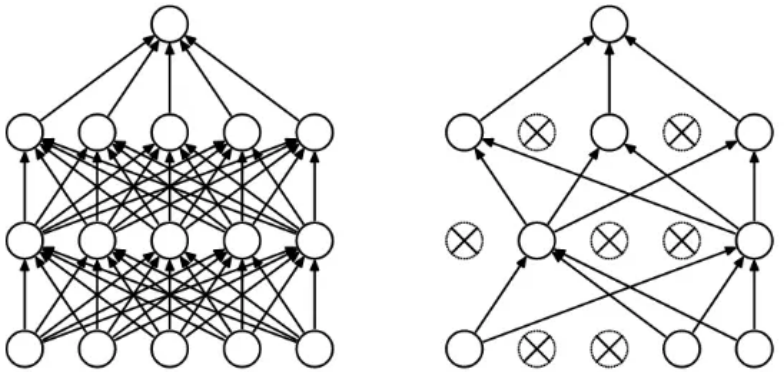




Trucos para mejorar los resultados

Dropout

- **¿Qué es el Dropout?**
  - Técnica para prevenir el sobreajuste.
  - "Apaga" aleatoriamente neuronas durante el entrenamiento.
- **Beneficios**:
  - Reduce la dependencia de neuronas individuales.
  - Mejora la generalización del modelo.
  - Convierte una red en una especie de comité de expertos



(a) Standard Neural Net (b) After applying dropout.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 15, 1929-1958. Recuperado de https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

Dropout

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(0.5)  # Dropout con probabilidad de 50%

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)  # Aplicar dropout después de la activación
        x = self.fc2(x)
        return x

# Crear una instancia de la red y un tensor de entrada
net = SimpleNet(10, 50, 2)
input_tensor = torch.randn(1, 10)
output = net(input_tensor)
```

Dropout

```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(32*28*28, 128)
        self.fc2 = nn.Linear(128, 10)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = x.view(x.size(0), -1)  # Aplanar
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Crear una instancia de la red y un tensor de entrada
net = SimpleCNN()
input_tensor = torch.randn(1, 1, 28, 28)
output = net(input_tensor)
```

Data Aumentation



Data Aumentation

```python
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader

# Definir las transformaciones para el aumento de datos
data_transforms = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),  # Volteo horizontal aleatorio
    transforms.RandomRotation(10),  # Rotación aleatoria de +-10 grados
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),  # Recorte aleatorio y cambio de tamaño
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),  # Cambios aleatorios en el color
    transforms.ToTensor(),  # Convertir PIL Image a tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # Normalización (valores estándar para modelos preentrenados)
])

# Cargar tu conjunto de datos con las transformaciones definidas
dataset = ImageFolder(root='path_to_your_images', transform=data_transforms)

# Crear un DataLoader
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Data Aumentation

1. **RandomHorizontalFlip**: Voltea horizontalmente la imagen con una probabilidad de 0.5.
2. **RandomRotation**: Rota la imagen aleatoriamente en un rango de +-10 grados.
3. **RandomResizedCrop**: Recorta una parte de la imagen y luego la cambia de tamaño a las dimensiones especificadas (en este caso, 224x224).
4. **ColorJitter**: Cambia aleatoriamente el brillo, el contraste, la saturación y el tono de la imagen.
5. **ToTensor**: Convierte la imagen (que está en formato PIL) a un tensor de PyTorch.
6. **Normalize**: Normaliza el tensor de la imagen con los valores de media y desviación estándar especificados (estos son valores estándar usados para modelos preentrenados en ImageNet).

NOTA: las transformaciones no crean físicamente nuevas imágenes almacenadas en tu disco. En lugar de eso, cuando una imagen es leída del conjunto de datos durante el entrenamiento (o cualquier otro proceso), las transformaciones se aplican "al vuelo". Esto significa que cada vez que la misma imagen es solicitada, es probable que se vea diferente debido a las transformaciones aleatorias que se le apliquen.

Transforming and augmenting images — Torchvision main documentation (pytorch.org)