

POSIBLES PREGUNTAS EXAMEN

1. ¿Qué significa que una imagen sea una matriz de valores y cómo se representan imágenes en escala de grises y en color?

Respuesta

En visión computacional, una imagen es una matriz numérica donde cada celda representa la intensidad de un píxel. La forma en la que se guardan esos valores depende del tipo de imagen:

Imágenes en escala de grises

- Se representan como una matriz 2D de tamaño (alto × ancho).
- Cada valor suele estar entre 0 y 255 (uint8):
 - 0 → negro
 - 255 → blanco
- Ejemplo: una imagen 200x200 tiene una matriz de 200×200 valores numéricos.

Imágenes en color (RGB)

- Se representan como un tensor 3D de dimensiones:
(alto × ancho × 3)
- Cada canal contiene intensidades independientes:
 - R → rojo
 - G → verde
 - B → azul
- La combinación de estos valores forma el color final del pixel.

En frameworks como PyTorch, el orden suele ser: (3 × alto × ancho).

2. ¿Qué es un kernel o filtro en visión computacional y cómo funciona una convolución?

Respuesta

Un kernel (o filtro) es una pequeña matriz numérica (por ejemplo 3×3 o 5×5) que se desliza sobre la imagen para producir una versión transformada de ella.

La convolución de una imagen es una operación matemática que aplica un filtro (kernel) a cada píxel de una imagen para crear una nueva imagen. Para ello, el filtro se desplaza por la imagen, multiplicando sus valores por los píxeles vecinos correspondientes y sumando los resultados para obtener el valor de cada píxel en la nueva imagen.

La convolución consiste en:

1. Ubicar el kernel sobre una región de la imagen.
2. Multiplicar cada elemento del kernel con su pixel correspondiente.
3. Sumar todos los productos.
4. Escribir ese valor en la salida.
5. Mover el kernel (según el stride) y repetir.

Fórmula simplificada:

$$output(i, j) = \sum_m \sum_n kernel(m, n) \cdot image(i + m, j + n)$$

¿Qué se puede lograr?

- Detección de bordes (Sobel, Prewitt, Laplace)
- Suavizado (blur, Gaussian blur)
- Realzar textura
- Operaciones morfológicas

El kernel determina qué patrón espacial se está realizando o suprimiendo.

3. ¿Qué representan los componentes de frecuencia baja y alta en una imagen?

Respuesta

La noción de frecuencia describe qué tan rápido cambian los valores de brillo en una imagen.

La frecuencia en imágenes es la tasa de cambio de los valores de intensidad de los píxeles a través de la imagen.

Frecuencias bajas

Cambios lentos y graduales en la intensidad de píxeles. Representan áreas uniformes o con gradientes suaves - Contienen la estructura general y formas básicas

- Áreas donde los valores cambian lentamente.
- Representan formas globales, iluminación y regiones suaves.
- Se preservan aplicando filtros *low pass* como el Gaussian blur.

Ejemplos: - Un cielo despejado sin nubes - Una pared pintada de un solo color - El fondo uniforme de una fotografía - Sombras suaves.

En términos técnicos: Si tomas píxeles vecinos, sus valores serán muy similares.

Ejemplo numérico: Píxeles adyacentes: [120, 122, 121, 123, 120] → Cambio lento (Baja frecuencia)

Frecuencias altas

Cambios rápidos y abruptos en la intensidad. Representan detalles finos y bordes nítidos. Son críticos para identificar límites de objetos

- Cambios bruscos en intensidades.
- Representan bordes, texturas, detalles finos.
- Se realzan aplicando filtros *high pass* (Sobel, Laplacian).

Ejemplos: - El borde entre un objeto negro y un fondo blanco - Texturas finas como la tela de una camisa - El cabello en un retrato - Líneas delgadas y detalles pequeños

En términos técnicos: Píxeles vecinos tienen valores muy diferentes.

Ejemplo numérico: Píxeles adyacentes: [50, 200, 45, 210, 40] → Cambio rápido (Alta frecuencia)

Importancia

La separación de frecuencias permite:

- Detección de bordes
- Reducción de ruido
- Segmentación
- Extracción de características

Las CNN aprenden automáticamente filtros que detectan frecuencias bajas en primeras capas y patrones complejos en capas posteriores.

3. ¿Qué es un histograma de imagen y qué información proporciona?

Respuesta

Un histograma es una gráfica que muestra la distribución de intensidades de una imagen.

En una imagen en escala de grises:

- Eje X: valores de intensidad (0–255)
- Eje Y: cuántos píxeles tienen esa intensidad

Información que ofrece

- Contraste general
- Iluminación predominante
- Presencia de regiones saturadas
- Distribución tonal (oscura, brillante, equilibrada)

Ejemplos

- Un histograma desplazado hacia la derecha → imagen muy clara.
- Un histograma concentrado en el centro → buen contraste.

- Un histograma bimodal → dos regiones claras diferenciadas (útil para umbralización).

4. Explica detalladamente cómo funciona el training loop y el testing loop en PyTorch. Incluye los pasos principales, el propósito de cada uno y las diferencias clave entre ambos.

Respuesta

1. PyTorch Training Loop (bucle de entrenamiento)

Este es el bucle donde ocurre el *aprendizaje*. Suele verse así de forma simplificada:

```

1. for epoch in range(epochs):
2.     model.train()
3.
4.     y_pred = model(X_train)      # 1. forward
5.     loss = loss_fn(y_pred, y_true) # 2. calcular pérdida
6.
7.     optimizer.zero_grad()       # 3. limpiar gradientes
8.     loss.backward()             # 4. backpropagation
9.     optimizer.step()           # 5. actualizar pesos

```

Ahora vamos línea por línea.

1.1 *for epoch in range(epochs):* – Recorrer epochs

- Una **epoch** = un recorrido completo sobre todo el conjunto de entrenamiento.
- **epochs** es el número de veces que el modelo verá **todo** el dataset.
- Durante cada epoch:
 - El modelo ve muchos *minibatches* de datos.
 - Ajusta sus pesos una y otra vez.

Idea clave: más epochs ≠ siempre mejor.

Demasiadas pueden llevar a **overfitting** (memoriza en lugar de generalizar).

1.2 *model.train()* – Poner el modelo en modo entrenamiento

PyTorch tiene dos modos importantes:

- *model.train()* → modo entrenamiento
- *model.eval()* → modo evaluación

Con *model.train()* pasan cosas internas:

- Capas como **Dropout** se activan (apagan neuronas aleatoriamente para regularización).
- **BatchNorm** calcula estadísticas (media y varianza) sobre el batch actual para normalizar.

Si **no** llamas a `model.train()`:

- El modelo podría comportarse como en evaluación.
- Dropout no se aplicaría.
- BatchNorm usaría estadísticas acumuladas, no las del batch → el entrenamiento puede volverse incorrecto.

1.3 `y_pred = model(X_train)` – Forward pass

Aquí:

- Estás pasando datos de entrada `X_train` al modelo.
- Internamente se ejecuta su método `forward()`:
 - Convoluciones
 - Activaciones
 - Pooling
 - Capas fully connected
 - etc.

Al final obtienes:

- `y_pred`: las **predicciones** del modelo para ese batch.
 - Pueden ser logits (valores sin softmax) o probabilidades, según la arquitectura y la loss.

1.4 `loss = loss_fn(y_pred, y_true)` – Calcular la pérdida

La **loss function** mide qué tan mal está el modelo:

- Para clasificación → típicamente CrossEntropyLoss.
- Para regresión → por ejemplo MSELoss o L1Loss.

Conceptualmente:

- Si `y_pred` se aleja de `y_true`, la pérdida es grande.

- Si y_{pred} se acerca a y_{true} , la pérdida es pequeña.

El entrenamiento consiste justamente en **hacer esta loss lo más pequeña posible** ajustando los pesos.

1.5 optimizer.zero_grad() – Limpiar gradientes acumulados

En PyTorch:

- Los gradientes **se acumulan** en cada parámetro.
- Esto permite técnicas avanzadas (por ejemplo, *gradient accumulation*), pero en el caso normal **no queremos acumularlos entre batches**.

Por eso antes de hacer `loss.backward()` siempre se hace:

`optimizer.zero_grad()`

Si te lo saltas:

- Los gradientes de diferentes batches se suman.
- El paso de actualización (`optimizer.step()`) usará gradientes incorrectos y el entrenamiento se descontrola.

1.6 loss.backward() – Backpropagation

Esta línea es el corazón del aprendizaje:

- PyTorch construye un grafo computacional durante el forward.
- `loss.backward()`:
 - Recorre ese grafo hacia atrás.
 - Aplica la regla de la cadena.
 - Calcula el gradiente de la loss respecto a cada parámetro con `requires_grad=True`.
- Esos gradientes quedan guardados en `param.grad` para cada parámetro del modelo.

Aún no se actualizan los pesos aquí, solo se calculan los gradientes.

1.7 optimizer.step() – Actualizar pesos

Finalmente:

`optimizer.step()`

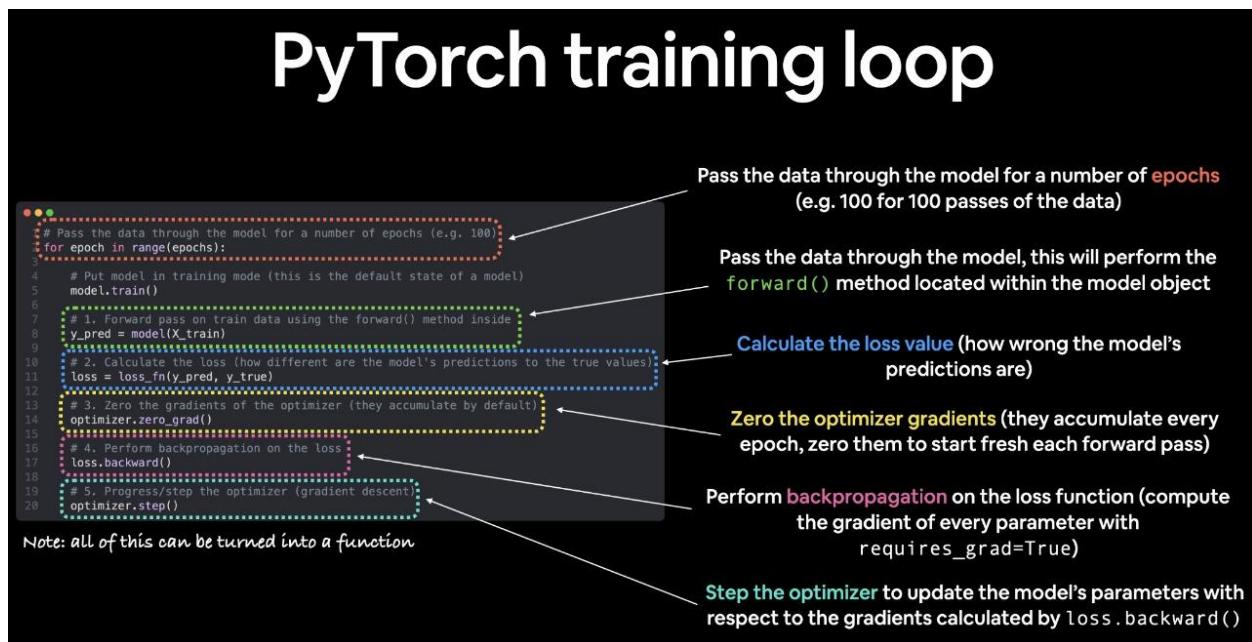
- El optimizador (SGD, Adam, etc.) lee los gradientes (param.grad) de cada parámetro.
- Actualiza los valores de cada peso siguiendo una **regla de actualización**:
 - Ejemplo en SGD:

$$w := w - \eta \cdot \frac{\partial L}{\partial w}$$

donde η es el learning rate.

Tras este paso:

- El modelo ha cambiado ligeramente para intentar reducir la loss.
- Repites esto muchas veces (epochs \times batches) → el modelo va “aprendiendo”.



2. PyTorch Testing Loop (bucle de evaluación / validación)

Este bucle **no entrena**, solo **mide rendimiento**.

Estructura típica:

```
1. model.eval()
2.
3. with torch.inference_mode():
4.     test_pred = model(X_test)           # 1. forward sin gradientes
5.     test_loss = loss_fn(test_pred, y_test) # 2. pérdida de test
```

En muchos casos se combina dentro de un loop sobre epochs, pero la idea es esta.

2.1 `model.eval()` – Modo evaluación

Cambia el comportamiento interno de ciertas capas:

- **Dropout** se desactiva (no apaga neuronas).
- **BatchNorm** usa estadísticas aprendidas (running mean/var), no las del batch.

¿Por qué es importante?

- Porque durante la evaluación queremos un comportamiento **determinista y estable**.
- No queremos ruido extra (como el de Dropout).
- Queremos ver el modelo tal como se usaría en producción.

Si evalúas sin `model.eval()`:

- Las métricas pueden ser inconsistentes.
- Puedes medir peores resultados de los reales.

2.2 `with torch.inference_mode()` – Desactivar gradientes

Este contexto:

```
1. with torch.inference_mode():
2.     ...
```

Hace dos cosas importantes:

1. **No se construye el grafo computacional**
 - No se guardan operaciones para backprop.
 - Ahorra memoria.
 - Acelera el cálculo.
2. Protege de errores
 - Asegura que no intentes llamar a `backward()` accidentalmente.

Diferencia con `torch.no_grad()`:

- Ambos desactivan el tracking de gradientes, pero inference_mode es aún más agresivo en optimizar memoria y velocidad para inferencia.

2.3 test_pred = model(X_test) – Forward sin entrenamiento

Similar al forward del training loop, pero:

- En **modo evaluación** (debido a `model.eval()`).
- Sin gradientes (por el contexto `inference_mode`).
- No hay dropout, ni actualizaciones de estadísticas de BatchNorm.

Obtienes las predicciones del modelo sobre el conjunto de prueba o validación.

2.4 test_loss = loss_fn(test_pred, y_test) – Pérdida de test

- Calculas la misma función de pérdida que en entrenamiento, pero ahora con datos de prueba.
- Esta loss no se usa para actualizar pesos, solo para medir qué tan bien generaliza el modelo.

Puedes además:

- Calcular accuracy, precision, recall, F1, etc.
- Guardar los valores en listas para graficar:
- `test_loss_values.append(test_loss.item())`

`epoch_count.append(epoch)`

The diagram shows a Python code snippet for a testing loop with annotations explaining its components:

```

1 # Setup empty lists to keep track of model progress
2 epoch_count = []
3 train_loss_values = []
4 test_loss_values = []
5
6 # Pass the data through the model for a number of epochs (e.g. 100) pochs:
7 for epoch in range(poachs):
8
9     ### Training loop code here ###
10
11     #### Testing starts ####
12     # Tell the model we want to evaluate rather than train (this
13     # turns off functionality used for training but not
14     # evaluation)
15     with torch.inference_mode():
16
17         # Turn on inference mode context manager
18         # 1. Forward pass on test data
19         test_pred = model(X_test)
20
21         # 2. Calculate loss on test data
22         test_loss = loss_fn(test_pred, y_test)
23
24         # Print what's happening every 10 epochs
25         if epoch % 10 == 0:
26             epoch_count.append(epoch)
27             train_loss_values.append(loss)
28             test_loss_values.append(test_loss)
29             print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test_loss}")
30
31 Note: all of this can be turned into a function

```

Annotations explain the code:

- Create empty lists for storing useful values (helpful for tracking model progress)
- Tell the model we want to evaluate rather than train (this turns off functionality used for training but not evaluation)
- Turn on `torch.inference_mode()` context manager to disable functionality such as gradient tracking for inference (gradient tracking not needed for inference)
- Pass the test data through the model (this will call the model's implemented `forward()` method)
- Calculate the test loss value (how wrong the model's predictions are on the test dataset, lower is better)
- Display information outputs for how the model is doing during training/testing every ~10 epochs (note: what gets printed out here can be adjusted for specific problems)

Note: all of this can be turned into a function

5. ¿Qué es la segmentación semántica y en qué se diferencia de la clasificación de imágenes?

Respuesta

Clasificación de imágenes

- Tarea global.
- El modelo asigna *una sola etiqueta* a toda la imagen.
- Ejemplo: "gato" o "perro".

Segmentación semántica

- Tarea pixel-a-pixel.
- El modelo etiqueta **cada pixel** con la clase correspondiente.
- Produce un mapa de colores que identifica regiones enteras.

Diferencias clave:

Clasificación	Segmentación
Una etiqueta por imagen	Una etiqueta por pixel
No sabe dónde está el objeto	Identifica la ubicación
Modelo simple (CNN tradicional)	Modelos como U-Net, DeepLab
No separa objetos	Puede separar límites

La segmentación es mucho más compleja, pero también más informativa.

6. Explica detalladamente en qué parte del flujo de entrenamiento de PyTorch se realiza el descenso de gradiente, cómo se crean tensores dentro de PyTorch y de qué maneras se puede actualizar la información contenida en un tensor. Proporciona ejemplos y la lógica interna que sigue PyTorch en cada paso.

En PyTorch, los tensores son la estructura fundamental para representar datos y parámetros de un modelo, mientras que el descenso de gradiente ocurre durante el proceso de entrenamiento con el fin de ajustar dichos parámetros. A continuación se explica cada concepto en detalle.

1. ¿En qué parte ocurre el descenso de gradiente en PyTorch?

El descenso de gradiente ocurre en dos pasos clave del training loop:

Paso 1 — loss.backward() → cálculo de gradientes

Cuando se calcula la pérdida:

```
1. loss = loss_fn(y_pred, y_true)
2. loss.backward()
```

PyTorch:

- Construye un grafo computacional durante el forward pass.
- Usa ese grafo para retropropagar la derivada de la pérdida con respecto a cada parámetro.
- Guarda los gradientes en param.grad para cada parámetro del modelo.

👉 Aquí NO se actualizan los pesos todavía.

Solo se calculan y almacenan los gradientes.

Paso 2 — optimizer.step() → actualización de parámetros (descenso de gradiente)

`optimizer.step()`

Este paso realiza el descenso de gradiente, es decir:

- Lee los valores almacenados en param.grad.
- Ajusta cada peso siguiendo la regla del optimizador elegido (SGD, Adam, RMSprop...).

Paso adicional — optimizer.zero_grad()

Antes del backward, siempre se limpian gradientes previos:

```
1. optimizer.zero_grad()
```

Esto evita que PyTorch acumule gradientes de diferentes batches.

2. ¿Cómo se crea un tensor en PyTorch?

PyTorch permite crear tensores de diferentes formas, similares a arrays de NumPy pero con soporte para GPU y gradientes.

2.1 Creación directa desde una lista

```
1. import torch  
2. t = torch.tensor([1, 2, 3])
```

2.2 Tensores aleatorios

```
1. x = torch.randn(3, 3)      # normal(0,1)  
2. y = torch.rand(2, 5)       # uniforme(0,1)
```

2.3 Tensores llenos

```
1. zeros = torch.zeros(4, 4)  
2. ones = torch.ones(2, 3)
```

2.4 Tensores con gradientes activados para entrenamiento

```
1. w = torch.randn(3, 3, requires_grad=True)
```

Esto es esencial para parámetros de un modelo, porque activa el cálculo automático de gradientes.

3. ¿Cómo actualizar información en un tensor?

Existen dos grandes formas de actualizar un tensor:

3.1 Actualización manual (cuando requires_grad=False)

Método A — Reasignación

```
1. t = torch.tensor([1,2,3])  
2. t = t + 1  
3. print(t)  # [2,3,4]
```

Método B — Operaciones *in-place*

```
1. t = torch.tensor([1,2,3])  
2. t.add_(5)  # operación in-place (modifica directamente)  
3. print(t)  # [6,7,8]
```

⚠ Notas importantes:

- Las operaciones con `_` modifican el tensor **in-place**.

- No es recomendable usar demasiadas operaciones in-place si requires_grad=True, porque pueden romper el grafo computacional.
-

3.2 Actualización automática durante el entrenamiento (cuando requires_grad=True)

Los tensores que representan **pesos del modelo** se actualizan durante el descenso de gradiente.

Esto ocurre exclusivamente al ejecutar:

```
1. optimizer.step()
```

Ejemplo:

```
1. w = torch.randn(3,3, requires_grad=True)
2. optimizer = torch.optim.SGD([w], lr=0.1)
3.
4. loss = (w.sum() - 10)**2
5. optimizer.zero_grad()
6. loss.backward()
7. optimizer.step()
```

Internamente:

- backward() calcula gradientes → w.grad
- optimizer.step() modifica w según esos gradientes
- w.grad se limpia en la siguiente iteración con zero_grad()

Resumen

- El descenso de gradiente en PyTorch ocurre en dos pasos:
 1. loss.backward() → cálculo de gradientes
 2. optimizer.step() → actualización de pesos
- Los tensores se crean con funciones como torch.tensor, torch.rand, torch.zeros, etc., y pueden tener requires_grad=True para participar en el entrenamiento.
- Los tensores pueden actualizarse manual o automáticamente:
 1. Manualmente mediante operaciones normales o *in-place*.
 2. Automáticamente mediante el optimizador durante el entrenamiento.

7. Explica cuáles son los rangos numéricos de los canales de los modelos de color RGB, HSV y HSL. Describe qué representa cada canal en cada modelo y menciona cómo varían sus rangos dependiendo de si la imagen está normalizada o en formato entero (uint8).

Los modelos de color RGB, HSV y HSL representan el color de manera distinta, por lo que cada uno utiliza canales con rangos numéricos diferentes. Estos rangos pueden expresarse en valores enteros (uint8) o valores normalizados entre 0 y 1, dependiendo del framework (OpenCV, PIL, PyTorch, NumPy, etc.).

1. Modelo RGB

El modelo RGB representa el color a través de la combinación de **tres intensidades**:

- **R:** rojo
- **G:** verde
- **B:** azul

Rangos típicos

Formato	Rango
uint8 (imagen normal)	0 a 255 para cada canal
float normalizado	0.0 a 1.0

Ejemplo

(255, 0, 128) # RGB en uint8

(1.0, 0.0, 0.5) # RGB normalizado

¿Qué representa?

- **0** → ausencia total del color.
- **255 o 1.0** → intensidad máxima.

RGB es el modelo más común para dispositivos (monitores, cámaras, etc.).

2. Modelo HSV (Hue, Saturation, Value)

El modelo HSV separa el color en componentes más interpretables:

- **H (Hue / Matiz):** el tipo de color (rojo, verde, azul, etc.).

- **S (Saturación):** intensidad del color (0 = gris).
- **V (Valor / Brillo):** luminosidad del color.

📌 *Rangos típicos en diferentes frameworks:*

Canal	OpenCV (uint8)	Normalizado	Significado
H	0 a 179	0.0 a 1.0	Ángulo en el círculo del color (0–360°, pero OpenCV lo divide a la mitad)
S	0 a 255	0.0 a 1.0	Qué tan “puro” es el color
V	0 a 255	0.0 a 1.0	Brillo o luminosidad

⚠ *Nota importante*

En **OpenCV**, el canal H no va de 0 a 360 porque lo guardan en 8 bits → ahorran espacio escalándolo a 0–179.

Pero en otros frameworks como PIL:

- **H va de 0 a 360**
- **S y V van de 0 a 1**

3. Modelo HSL (Hue, Saturation, Lightness)

HSL es similar a HSV, pero la tercera componente representa **luminosidad**, no brillo.

- **H (Hue / Matiz)**
- **S (Saturación)**
- **L (Lightness / Luminosidad)**

📌 *Rangos típicos*

Canal	Rango
H	0 a 360 (normalizado 0.0–1.0)
S	0 a 1
L	0 a 1

En HSL:

- $L = 0 \rightarrow$ negro
- $L = 1 \rightarrow$ blanco
- $L = 0.5 \rightarrow$ color puro con saturación máxima

Es un modelo usado más en diseño gráfico y web (CSS), menos en visión computacional.

Comparación general de rangos

Modelo	Canal	Rango típico	Notas
RGB	R, G, B	0–255 o 0–1	Intensidad de cada color primario
HSV	H	0–179 (OpenCV) o 0–360	Matiz del color
	S, V	0–255 o 0–1	Saturación y brillo
HSL	H	0–360	Matiz
	S, L	0–1	Saturación y luminosidad

Nota

- ◆ Value (HSV) mide la *cantidad de luz que entra al color* (máximo de RGB).
- ◆ Lightness (HSL) mide *qué tan claro u oscuro es el color en general* (promedio entre max y min).

8. Explica el papel del padding, el stride y las operaciones de pooling en las redes convolucionales.

1. Padding

El *padding* consiste en agregar bordes extras alrededor de la imagen, normalmente llenos con ceros.

¿Por qué es necesario el padding?

Sin padding:

- Una convolución de tamaño $k \times k$ reduce el tamaño de la imagen.
Ej.: una imagen 32×32 con un kernel 3×3 produce una salida de 30×30.

- Los bordes se procesan menos que los píxeles centrales.
- La imagen pierde información espacial en cada capa.

Beneficios del padding:

1. Preserva el tamaño espacial

Con *padding* adecuado (por ejemplo, 1 para un kernel 3×3), la salida queda del mismo tamaño que la entrada (esto se llama *same padding*).

2. Permite más capas convolucionales sin colapsar la imagen

Sin padding, una red profunda se queda sin dimensiones rápidamente.

3. Permite que la convolución “vea” los bordes

Evita que los bordes se pierdan durante el procesamiento.

Fórmula general para el tamaño de salida:

$$O = \frac{W - K + 2P}{S} + 1$$

donde:

- W = tamaño de entrada
- K = tamaño del kernel
- P = padding
- S = stride

Tipos de padding

- **padding="valid"**
 - Significa “sin padding”.
 - Se utilizan solo ubicaciones donde el kernel cabe completamente dentro de la imagen.
 - Produce una salida más pequeña que la entrada.
 - Es el comportamiento por defecto si no se especifica nada.
- **padding="same"**
 - Significa “aplicar padding para que la salida tenga el mismo alto y ancho que la entrada”.
 - El sistema calcula automáticamente cuántos píxeles deben añadirse alrededor.

- Común con kernel $3 \times 3 \rightarrow$ padding de 1 en cada lado.

2. Stride

El *stride* indica cuántos píxeles se avanza el kernel en cada paso durante la convolución.

Efecto del stride:

- **Stride = 1**
 - Convolución estándar.
 - Mapa de salida detallado, alta resolución espacial.
- **Stride > 1**
 - Reduce el tamaño de la salida (downsampling).
 - Hace la convolución más eficiente.
 - Reduce el detalle espacial.

Ejemplo:

- Entrada: 32×32
- Kernel: 3×3
- Padding: 1
- Stride = 2

$$O = \frac{32 - 3 + 2(1)}{2} + 1 = 16$$

Reduces la resolución **a la mitad**.

3. Pooling

El *pooling* es una operación que **reduce la dimensionalidad espacial** tomando valores representativos dentro de una ventana (por ejemplo 2×2).

Tipos comunes:

- **Max pooling:** toma el valor máximo.
- **Average pooling:** calcula el promedio.

¿Para qué sirve el pooling?

1. Reduce el tamaño espacial

Ejemplo: kernel 2×2 con stride 2 → reduce la imagen a la mitad en cada dimensión.

2. Mantiene las características importantes

- Max pooling conserva bordes y activaciones fuertes.

- Average pooling suaviza la representación.

3. Reduce sobreajuste (overfitting)

Al eliminar detalles excesivos.

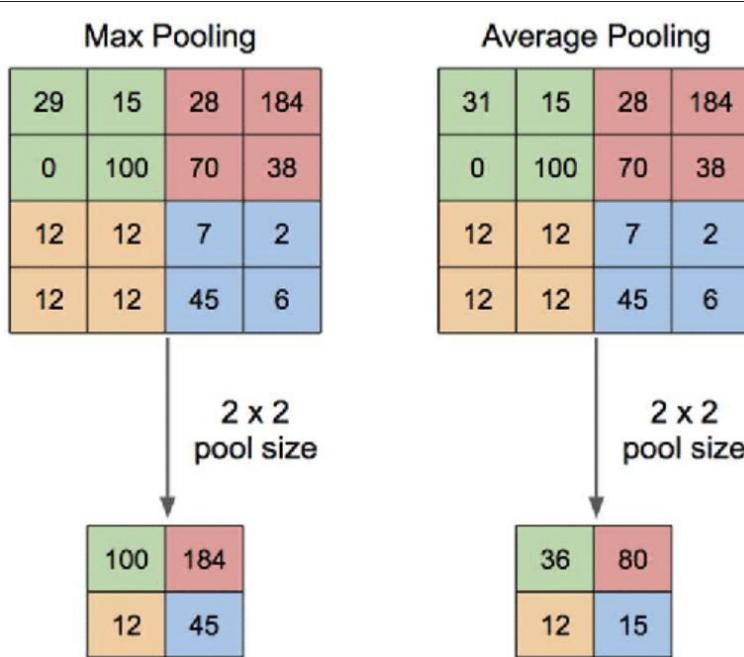
4. Aumenta la invariancia traslacional

La red se vuelve más robusta a movimientos pequeños del objeto en la imagen.

Ejemplo:

Mapa de 32×32 → pooling 2×2 , stride 2 → salida 16×16 .

Sin pooling, una CNN necesitaría muchas más capas para capturar patrones robustos.



9. Explica detalladamente la diferencia entre *feature extraction* y *fine-tuning* en el contexto de *transfer learning*.

En visión computacional, el *transfer learning* consiste en utilizar un modelo preentrenado (por ejemplo ResNet, VGG, EfficientNet entrenados en ImageNet) para resolver una nueva tarea. Existen dos estrategias principales para aprovechar estos modelos: feature extraction y fine-tuning. Aunque comparten la misma idea general, su funcionamiento y comportamiento son significativamente diferentes.

1. Feature Extraction

En este enfoque, el modelo preentrenado se usa como **extractor de características fijo**.

¿Qué significa “fijo”?

- **Todas las capas convolucionales están congeladas.**
Es decir, sus pesos **NO se actualizan** durante el entrenamiento.
- Solo se entrena una nueva capa final (generalmente una capa totalmente conectada) que se ajusta a la tarea nueva.

¿Qué partes se entrenan?

- **✗ Capas convolucionales → no se entrenan**
- **✓ Clasificador final → sí se entrena**

Ejemplo en pseudocódigo:

```
for param in model.features.parameters():
    param.requires_grad = False
```

¿Qué ventaja tiene?

- **Entrenamiento muy rápido**, porque la mayoría del modelo no se actualiza.
- Requiere **pocos datos**.
- Evita sobreajuste cuando el dataset es pequeño.

¿Qué desventaja tiene?

- El modelo depende totalmente de las características aprendidas para ImageNet.
- Si la nueva tarea es muy distinta del dataset original, el rendimiento puede ser limitado.

¿Cuándo usar Feature Extraction?

- Cuando tienes **pocos datos** (< 2,000 imágenes).
- Cuando tu tarea es similar a ImageNet (gatos vs perros, vehículos, animales, etc.).

- Cuando necesitas entrenamiento rápido y eficiente.

● 2. Fine-Tuning

En este enfoque, el modelo preentrenado **sí se modifica** parcialmente o totalmente para adaptarse a la nueva tarea.

¿Qué significa “ajustar finamente”?

- Se **descongelan** algunas o todas las capas del modelo.
- Se vuelven a entrenar con un learning rate bajo para no destruir lo aprendido.

¿Qué partes se entrena?

- ✓ Capas superiores del modelo (las últimas convoluciones)
- ✓ Clasificador final
- (Opcional) ✓ Todas las capas del modelo

Ejemplo:

```
1. for param in model.parameters():
2.     param.requires_grad = True # descongelar todo
```

O solo algunas capas:

```
1. for param in model.layer4.parameters():
2.     param.requires_grad = True
```

¿Qué ventaja tiene?

- El modelo aprende **representaciones específicas para el nuevo dataset**.
- Suele ofrecer **mayor precisión** que feature extraction.

¿Qué desventaja tiene?

- Requiere **más datos** para evitar sobreajuste.
- El entrenamiento es **más lento** y costoso computacionalmente.
- Un learning rate inapropiado puede destruir los pesos preentrenados.

¿Cuándo usar Fine-Tuning?

- Cuando tienes un dataset **más grande** (> 5,000 imágenes).
- Cuando la tarea es **muy diferente** a ImageNet (microscopía, radiografías, satélite).

- Cuando necesitas el mejor rendimiento posible.

9. ¿Cuál es el objetivo principal de un autoencoder y cuáles son sus componentes fundamentales?

Respuesta

Un autoencoder es una red neuronal diseñada para aprender a copiar sus entradas, pero con una condición importante: antes de intentar reconstruir la entrada, el modelo está obligado a comprimirla en una versión más pequeña. Esta compresión obliga a la red a aprender qué es realmente importante de los datos y qué es redundante.

Un autoencoder es una arquitectura de red neuronal diseñada para aprender una representación comprimida de los datos, de manera no supervisada. Su objetivo no es clasificar ni predecir etiquetas, sino reconstruir la entrada a partir de una versión compacta de ella misma.

Un autoencoder siempre tiene tres partes que trabajan juntas: el **codificador**, el **espacio latente** y el **decodificador**.

1. El Codificador (Encoder)

El codificador es la parte de la red que recibe la entrada original, por ejemplo una imagen. Su trabajo es **comprimirla**.

Si la imagen tiene 784 valores (como en MNIST), el codificador la reduce poco a poco:

784 → 256 → 64 → 16

En cada paso, aprende cuáles valores importan más y cuáles puede descartar sin perder demasiada información.

El codificador se encarga de decir:

“Voy a resumir este dato grande en algo pequeño pero significativo.”

2. El Espacio Latente (Latent Space)

El espacio latente es el **resultado final de la compresión**.

Es un vector muy pequeño (por ejemplo de 8, 16 o 32 valores) que contiene la información más esencial de la entrada original.

Aquí es donde está la “versión comprimida” del dato.

Si el autoencoder funciona bien, este vector latente contiene:

- lo más importante del dato,
- lo más característico,
- lo que define su identidad.

Es como la **descripción mínima necesaria** para poder reconstruir el dato después.

Aquí está el corazón del autoencoder:

si la compresión es buena, la reconstrucción también será buena.

3. El Decodificador (Decoder)

El decodificador recibe el vector comprimido (el espacio latente) y trata de reconstruir la entrada original.

Su trabajo es como usar el resumen que hiciste para recrear la foto, intentando que se parezca lo más posible a la original:

$16 \rightarrow 64 \rightarrow 256 \rightarrow 784$

El decodificador aprende a “**descomprimir**” la representación latente.

Si la reconstrucción sale parecida a la entrada original, significa que:

- el codificador hizo un buen resumen
- y el decodificador supo interpretar ese resumen

Toda la red mejora mediante entrenamiento reduciendo la diferencia entre la entrada y la salida reconstruida.

Resumen corto para examen

- *Objetivo del autoencoder:*
Aprender una representación comprimida y reconstruir la entrada lo mejor posible.
- *Encoder:*
Comprime la entrada extrayendo características importantes.
- *Latent Space:*
Representación comprimida del dato; punto de estrangulación (bottleneck).
- *Decoder:*
Reconstruye la entrada a partir del vector latente.

👉 En conjunto, el autoencoder aprende **una codificación significativa** del dato original sin necesidad de etiquetas.

10. ¿Qué es el data augmentation en visión computacional y cuál es su propósito?

Respuesta

Data augmentation es una técnica utilizada en visión computacional para aumentar artificialmente el tamaño y la diversidad del dataset sin necesidad de recolectar nuevas imágenes.

La idea es tomar las imágenes originales y modificarlas de formas controladas, como rotarlas, voltearlas, cambiar el brillo o recortarlas, creando nuevas versiones que siguen representando al mismo objeto. Estas imágenes adicionales no son “nuevas” en el sentido estricto, pero sí aportan variaciones útiles.

¿Para qué sirve el data augmentation?

El objetivo principal del data augmentation es hacer que el modelo sea más robusto y generalice mejor.

Cuando un modelo solo ve imágenes muy similares entre sí, tiende a memorizar los datos (overfitting) y falla con imágenes nuevas.

Al aplicar augmentations, el modelo se entrena con versiones variadas de las imágenes, lo que le enseña a reconocer el objeto sin importar cambios pequeños, como:

- si está ligeramente rotado,
- si está más oscuro,
- si está un poco alejado,
- si está volteado,
- si tiene ruido, etc.

En otras palabras:

👉 El data augmentation enseña al modelo a no depender de detalles insignificantes y a reconocer el patrón real del objeto.

Ejemplos comunes de data augmentation

Los augmentations más típicos en visión computacional incluyen:

- *Rotaciones* (15°, 30°, etc.)
- *Volteo horizontal* (flip horizontal)

- *Recortes aleatorios* (random crops)
- *Cambio de brillo o contraste*
- *Zoom aleatorio*
- *Desplazamientos* (translations)
- *Agregar ruido leve*
- *Cambios de color* (color jitter)

11. Supongamos que queremos construir un sistema de visión por computadora que prediga la ubicación de uno o varios “puntos azules” dentro de una imagen. Explica qué datos serían necesarios para entrenar el modelo, cómo debería estructurarse el tensor de entrada para la red neuronal y cómo debería ser la capa de salida del modelo para poder predecir correctamente las coordenadas de los puntos azules.

Respuesta

Para entrenar un sistema que prediga la ubicación de “puntos azules” en una imagen, necesitamos tres elementos clave: **los datos adecuados, una representación correcta para la red neuronal y una capa de salida apropiada para predecir coordenadas.**

1. Datos necesarios para entrenar un sistema que prediga puntos azules

Lo primero y más importante es tener un conjunto de datos formado por:

a) *Imágenes de entrada*

- Cada imagen debe mostrar el escenario donde aparecen los puntos azules.
- Pueden ser fotos, capturas de pantalla, frames de video, etc.

b) *Etiquetas numéricas con las coordenadas de cada punto azul*

- Estas coordenadas suelen representarse como **(x, y)** dentro del espacio de la imagen.
- Si solo existe un punto azul por imagen, cada etiqueta es:

$$[x, y]$$

- Si hay varios puntos azules (por ejemplo 3 puntos), la etiqueta puede ser:

$$[x_1, y_1, x_2, y_2, x_3, y_3]$$

Para obtener estas etiquetas es necesario:

- Anotar manualmente dónde está cada punto,
- o usar un sistema automático de detección previo.

c) Normalización de coordenadas (opcional pero recomendado)

Las coordenadas suelen normalizarse entre 0 y 1 dividiendo entre el ancho y alto de la imagen:

$$x_{norm} = \frac{x}{W}, y_{norm} = \frac{y}{H}$$

Esto ayuda a entrenar más establemente.

2. ¿Cómo sería el tensor de entrada para la red neuronal?

En visión computacional, las imágenes se representan como tensores con tamaño:

$$(canales, alto, ancho)$$

Ejemplos:

Imagen RGB

$$3 \times H \times W$$

Imagen en escala de grises

$$1 \times H \times W$$

Ejemplo concreto

Si la imagen es 256×256 RGB, el tensor de entrada sería:

$$(3, 256, 256)$$

Este tensor se alimenta a la CNN para extraer características.

3. ¿Cómo debe ser la última capa del modelo para predecir puntos azules?

Como lo que queremos es predecir coordenadas numéricas, la salida no debe ser una softmax ni una clasificación.

Debe ser una capa totalmente conectada (Linear / Dense) cuya cantidad de neuronas dependa del número de puntos que queremos predecir.

Si hay un solo punto azul por imagen

La salida debe tener 2 valores:

$$(x, y)$$

Ejemplo en PyTorch:

```
1. self.fc_out = nn.Linear(128, 2)
```

Si hay N puntos azules

La salida debe tener:

$$2N \text{ valores}$$

Por ejemplo, para 3 puntos:

```
1. self.fc_out = nn.Linear(128, 6)
```

Función de activación en la capa final

Generalmente:

- **No se usa softmax** (porque no es clasificación).
- Se utiliza una **capa lineal pura** que produce valores reales.
- Si las coordenadas están normalizadas, se puede usar **sigmoid** para forzar salida en [0,1].

Función de pérdida adecuada

Como es un problema de regresión:

- **MSELoss**

- **SmoothL1Loss**
son las más utilizadas.

12. Supongamos que queremos entrenar un modelo de visión por computadora que determine si en una imagen hay o no hay un “punto azul”. Explica qué datos serían necesarios, cómo se representaría el tensor de entrada para la red neuronal y cómo debería ser la última capa del modelo para realizar una clasificación binaria correcta.

Respuesta

Para resolver un problema donde el objetivo es **clasificar** si un punto azul existe o no existe en una imagen, se necesita preparar adecuadamente tanto los datos como el modelo. A diferencia del caso donde predecimos coordenadas (regresión), aquí solo queremos una respuesta **sí o no**, por lo que la salida es mucho más simple.

1. Datos necesarios para entrenar la clasificación binaria

Se necesitan dos tipos de información:

a) *Imágenes de entrada*

Estas son las imágenes donde podría o no aparecer el punto azul.

Pueden contener:

- un punto azul en cualquier posición, o
- ningún punto azul en absoluto.

b) *Etiquetas binarias*

Cada imagen debe tener una etiqueta numérica indicando si el punto aparece:

- **1** → la imagen **sí** contiene un punto azul
- **0** → la imagen **no** contiene un punto azul

A diferencia de la versión anterior del problema, **no necesitamos las coordenadas** del punto.

Aquí solo se requiere saber si el punto está o no está.

2. ¿Cómo sería el tensor de entrada para la red neuronal?

Igual que en cualquier tarea de visión con redes neuronales, las imágenes deben convertirse a tensores con tamaño:

$$(C, H, W)$$

Donde:

- **C** = número de canales (1 si es gris, 3 si es RGB)
- **H** = alto
- **W** = ancho

Ejemplos:

Imagen RGB 256×256:

$$(3, 256, 256)$$

Imagen en escala de grises 128×128:

$$(1, 128, 128)$$

Estos tensores se entregan a una CNN que extrae características relevantes y luego decide si hay punto azul.

3. ¿Cómo debe ser la última capa del modelo para clasificación binaria?

Como esto es una clasificación de dos opciones (sí/no), la arquitectura final debe estar diseñada para producir **probabilidades** de pertenecer a una de las dos clases.

Hay dos formas comunes de estructurar la salida:

Opción A – 1 neurona con activación sigmoid (la más típica)

La red termina con una sola neurona que produce un valor entre 0 y 1.

Ejemplo de capa final:

```
1. self.fc_out = nn.Linear(128, 1)
```

Y luego se aplica:

```
1. output = torch.sigmoid(self.fc_out(x))
```

Interpretación:

- output $\approx 0 \rightarrow$ **no hay punto azul**
- output $\approx 1 \rightarrow$ **sí hay punto azul**

Función de pérdida recomendada:

- **Binary Cross-Entropy Loss** \rightarrow nn.BCELoss() o nn.BCEWithLogitsLoss()
(La versión *with logits* es más estable y se usa con salida sin sigmoid.)

Opción B – 2 neuronas con softmax (menos común pero válido)

La red produce dos valores:

- uno para "no existe punto azul",
- otro para "sí existe punto azul".

Capa final:

```
1. self.fc_out = nn.Linear(128, 2)
```

Luego se aplica softmax para obtener probabilidades:

```
1. probs = torch.softmax(logits, dim=1)
```

Función de pérdida:

- **CrossEntropyLoss**

13. Explica qué función de activación es apropiada para la última capa en los siguientes casos: clasificación binaria, clasificación multiclase, multietiqueta y problemas de regresión. Justifica cada elección.

Respuesta

1. Clasificación binaria (sí / no)

Etiquetas esperadas

- 0 = clase negativa
- 1 = clase positiva

Función de activación recomendada

→ **Sigmoid**

La sigmoid produce un valor entre 0 y 1 que puede interpretarse como probabilidad.

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

La salida ideal de la red es un número muy cercano a:

- 0 si la clase es negativa
- 1 si la clase es positiva

Pérdida adecuada

- BCEWithLogitsLoss (sin sigmoid manual)
- o BCELoss (si ya aplicaste sigmoid)

2. Clasificación multiclas (solo una clase correcta de varias)

Etiquetas esperadas

Normalmente:

- una sola clase correcta por imagen
- representada como un índice entero: 0, 1, 2, ..., K-1

Función de activación recomendada

→ **Softmax**

El softmax convierte un vector en probabilidades que suman 1.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Se usa cuando solo una clase puede ser verdadera.

Pérdida adecuada

- CrossEntropyLoss (que ya incluye softmax internamente)

3. Clasificación multietiqueta (puede tener varias clases al mismo tiempo)

Ejemplos:

- en una imagen puede haber “perro + césped + persona”
- etiquetas tipo: [1, 0, 1, 1, 0]

Función de activación recomendada

→ **Sigmoid por cada neurona de salida**

Por qué:

- Cada etiqueta es independiente.
- No queremos que las probabilidades sumen 1 (como con softmax).
- Cada salida debe estar entre 0 y 1.

Pérdida adecuada

- BCEWithLogitsLoss aplicado a cada salida

4. Regresión (predecir valores numéricos)

Ejemplos:

- predecir coordenadas de puntos
- estimar la edad
- estimar una distancia

Función de activación recomendada

→ **Ninguna (salida lineal)**

La red debe poder producir valores reales, positivos o negativos, sin restricciones.

Excepción: si las salidas deben estar en un rango específico:

- coordenadas normalizadas: usar **sigmoid**
- valores de 0 a 100: usar **ReLU capped**, o normalizar los datos

Pérdidas típicas

- MSELoss
- SmoothL1Loss

Resumen final (para memorizar)

Tipo de problema	Activación final	# neuronas de salida	Pérdida

Binario	Sigmoid	1	BCE / BCEWithLogits
Multiclas (una sola etiqueta)	Softmax	N clases	CrossEntropyLoss
Multietiqueta	Sigmoid (por etiqueta)	N etiquetas	BCEWithLogits
Regresión	Sin activación (lineal)	depende del problema	MSE / SmoothL1

14. Dada la siguiente matriz de entrada (imagen en escala de grises):

$$X = \begin{bmatrix} 1 & 2 & 0 & 1 \\ 3 & 1 & 2 & 2 \\ 1 & 0 & 1 & 3 \\ 2 & 1 & 0 & 1 \end{bmatrix}$$

y el siguiente filtro (kernel) de tamaño 2×2:

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

Aplica una convolución sobre X usando:

- padding = 0 (valid)
 - stride = 1
1. Calcula el tamaño de la salida.
 2. Aplica la convolución manualmente y escribe toda la matriz resultante.
 3. Explica brevemente por qué el tamaño de salida cambia según el padding y el stride.

1. Tamaño de la salida

Usamos la fórmula general:

$$O = \frac{(W - K + 2P)}{S} + 1$$

Donde:

- $W = 4$ (tamaño de la imagen)
- $K = 2$ (tamaño del kernel)
- $P = 0$ (padding = valid)
- $S = 1$ (stride)

Entonces:

$$O = \frac{(4 - 2 + 0)}{1} + 1 = 3$$

La salida será de tamaño 3x3.

● 2. Aplicación manual de la convolución

Vamos a deslizar el kernel sobre la imagen, multiplicar elemento por elemento y sumar.

Posición (1,1)

Ventana:

$$\begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix}$$

Multiplicamos por el kernel:

$$1(1) + 2(0) + 3(-1) + 1(1) = 1 + 0 - 3 + 1 = -1$$

Posición (1,2)

Ventana:

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

$$2(1) + 0(0) + 1(-1) + 2(1) = 2 + 0 - 1 + 2 = 3$$

Posición (1,3)

Ventana:

$$\begin{bmatrix} 0 & 1 \\ 2 & 2 \end{bmatrix}$$
$$0(1) + 1(0) + 2(-1) + 2(1)$$
$$= 0 + 0 - 2 + 2 = 0$$

Primera fila de salida:

$$[-1 \quad 3 \quad 0]$$

Posición (2,1)

Ventana:

$$\begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix}$$
$$3(1) + 1(0) + 1(-1) + 0(1)$$
$$= 3 + 0 - 1 + 0 = 2$$

Posición (2,2)

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$$
$$1(1) + 2(0) + 0(-1) + 1(1)$$
$$= 1 + 0 + 0 + 1 = 2$$

Posición (2,3)

$$\begin{bmatrix} 2 & 2 \\ 1 & 3 \end{bmatrix}$$
$$2(1) + 2(0) + 1(-1) + 3(1)$$
$$= 2 + 0 - 1 + 3 = 4$$

Segunda fila de salida:

$$[2 \quad 2 \quad 4]$$

Posición (3,1)

$$\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$$
$$1(1) + 0(0) + 2(-1) + 1(1) = 1 + 0 - 2 + 1 = 0$$

Posición (3,2)

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$
$$0(1) + 1(0) + 1(-1) + 0(1)$$
$$= 0 + 0 - 1 + 0 = -1$$

Posición (3,3)

$$\begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix}$$
$$1(1) + 3(0) + 0(-1) + 1(1)$$
$$= 1 + 0 + 0 + 1 = 2$$

Tercera fila de salida:

$$[0 \quad -1 \quad 2]$$

3. Matriz final resultante

$$\boxed{\begin{bmatrix} -1 & 3 & 0 \\ 2 & 2 & 4 \\ 0 & -1 & 2 \end{bmatrix}}$$

4. Breve explicación del efecto del padding y el stride

- Con **padding = valid (0)**, el kernel solo se coloca donde cabe completamente → la salida se vuelve más pequeña.
- Con **padding = same**, se agregan bordes para conservar el tamaño original.
- Con **stride = 1**, el kernel avanza pixel por pixel → salida grande.
- Con **stride > 1**, el kernel da “saltos” más grandes → salida más pequeña.

15. Una red convolucional recibe una imagen de entrada de tamaño:

Imagen de entrada: $3 \times 64 \times 64$

La red tiene las siguientes capas en orden:

1. Conv2D

- filtros: 16
- kernel: 3×3
- stride: 1
- padding: 1 ("same")

2. MaxPooling

- kernel: 2×2
- stride: 2

3. Conv2D

- filtros: 32
- kernel: 3×3
- stride: 1
- padding: 1 ("same")

4. MaxPooling

- kernel: 2×2
- stride: 2

Después de estas capas convolucionales + pooling, queremos conectar la salida a una fully connected layer.

Pregunta

¿Cuál es el tamaño final del mapa de características antes de entrar a la capa fully connected, y cuántas neuronas recibe exactamente la capa FC?

Paso 1: Entrada

$$3 \times 64 \times 64$$

Paso 2: Primera Conv2D (16 filtros, 3x3, stride 1, padding 1)

Como el padding es “same”, el tamaño espacial se conserva.

$$\text{Salida} = 16 \times 64 \times 64$$

Paso 3: MaxPooling (2x2, stride 2)

El pooling reduce a la mitad altura y anchura:

$$64 \rightarrow 32$$

$$O = \frac{(64 - 2 + 2 * 0)}{2} + 1$$

$$\text{Salida} = 16 \times 32 \times 32$$

Paso 4: Segunda Conv2D (32 filtros, 3x3, stride 1, padding 1)

Padding "same" otra vez → tamaño espacial se conserva.

$$\text{Salida} = 32 \times 32 \times 32$$

Paso 5: MaxPooling (2x2, stride 2)

Reduce a la mitad otra vez:

$$32 \rightarrow 16$$

$$O = \frac{(32 - 2 + 2 * 0)}{2} + 1$$

$$\text{Salida final} = 32 \times 16 \times 16$$

Resultado final del mapa antes de la fully connected

$$32 \times 16 \times 16$$

Este tensor es la entrada directa para la FC.

Paso 6: ¿Cuántas neuronas recibe la fully connected?

Antes de entrar a una capa totalmente conectada, el tensor debe aplanarse (flatten).

$$32 \times 16 \times 16 = 32 \times 256 = 8192$$

Entonces:

La capa FC recibe exactamente 8192 valores como entrada.

PREGUNTAS EXTRAS

¿Quién inventó las redes convolucionales?

- El principal desarrollador de las redes neuronales convolucionales (CNN) es Yann LeCun, quien introdujo la arquitectura en 1989.

¿Quién es el abuelo de la IA moderna?

- Geoffrey Hinton