Quanvolutional Convolutional Neural Network

Universidad Nacional Autónoma de México
Laboratorio Avanzado de Procesamiento de Imágenes
Académicos: Dr. Boris Escalante Ramírez
Autor: Sebastián González Juárez

Resumen.
En este proyecto se comparó una QCNN con una CNN clásica para la clasificación de PneumoniaMNIST. Ambas mostraron curvas de entrenamiento estables y sin sobreajuste, con trayectorias casi idénticas en pérdida y accuracy. Sin embargo, en el conjunto de prueba la QCNN obtuvo métricas ligeramente superiores en los indicadores más relevantes para aplicaciones médicas, especialmente recall y F1-score, reduciendo falsos negativos y manteniendo una precisión comparable. Los mapas cuánticos producidos por los cuatro qubits mostraron patrones distintos y coherentes, y la evolución suave de los parámetros del ansatz confirmó un entrenamiento estable. En conjunto, la QCNN ofrece un rendimiento similar al de una CNN clásica, pero con una ventaja consistente en métricas clínicas clave y con un potencial prometedor para explorar ansatz más expresivos en futuros trabajos.

Introducción.

Objetivo.

Tras el éxito logrado en la carpeta donde encontramos el proyecto "PneumoniaMNIST_with_Quanvolution", el cual busco la implementación de imágenes más grandes al llevar las Quanvoluciones modelo. El objetivo de esta colección de 3 códigos fue poder realizar las simulación en un entorno que nos permita llevarlo al GPU y así poder hacer más escalable el proyecto, como implementarlo junto a una CNN.

- **"GPU_PneumoniaMNIST_with_Quanvolution.ipynb"**

  Donde se da este primer paso de hacer uso del GPU y lograr acelerar el funcionamiento de las Quanvoluciones corriendo varias a la vez.

- **"Quanvolution_CNN_PneumoniaMNIST.ipynb"**

  Escalar el proyecto para ver que sucede al implementarle una CNN. Esto con el fin de en un futuro poder utilizar el modelo hibrido para conjuntos de imágenes más grandes. Fue donde construí mi primer QCNN.

- **"Quanvolution_3×3_with_lecture_in_1_qubit.ipynb"**

  Se llevo a cabo una mejora al Ansatz y para ello se requería de un modelo que realizará la Quanvolución con 9 qubits.

El presente documento busca resumir la idea de construcción de este tipo de redes hibridas, me centro detalladamente en explicar el segundo código: **"Quanvolution_CNN_PneumoniaMNIST.ipynb".**

Algoritmos cuánticos variacionales (VQA).

Un VQA combina un circuito cuántico parametrizado con un optimizador clásico para minimizar una función de costo, dando lugar a un esquema híbrido dividido en cuatro bloques fundamentales:

(1) **Feature Map:**
Es la etapa encargada de codificar datos clásicos en estados cuánticos mediante compuertas que preparan superposiciones en los qubits.

(2) **Variational Quantum Circuit (VQC) o Ansatz:**
Corresponde al núcleo del modelo cuántico; aquí se aplican compuertas dependientes de parámetros entrenables y compuertas de entrelazamiento, repitiéndose en capas para aumentar la capacidad expresiva.

(3) **Measurement:**
Al finalizar el circuito se realizan mediciones que devuelven valores clásicos (típicamente expectativas de observables), los cuales alimentan la función de costo.

(4) **Classical Model:**
Finalmente, un optimizador clásico ajusta los parámetros del VQC, y en aplicaciones prácticas suele integrarse con una red neuronal clásica que termina la tarea de aprendizaje supervisado.

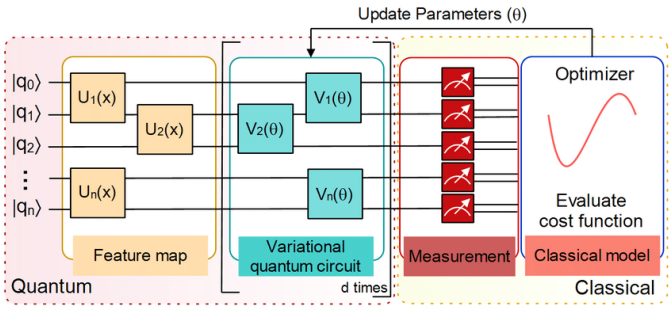


Ilustración 1. Algoritmo cuántico variacional.

Redes convolucionales (CNN).

Una Convolutional Neural Network (CNN) es un tipo de red neuronal diseñada para procesar datos con estructura espacial, como imágenes, utilizando filtros convolucionales que detectan patrones locales (bordes, texturas, formas) de manera jerárquica. A través de capas de convolución, activación y pooling, la CNN aprende representaciones cada vez más abstractas que permiten clasificar o analizar imágenes con gran eficiencia.

Quantum Convolutional Neural Network (QCNN).

Una **Quantum Convolutional Neural Network (QCNN)** es una arquitectura híbrida que combina ideas de las redes convolucionales clásicas con circuitos cuánticos variacionales. En lugar de usar filtros convolucionales tradicionales, una QCNN aplica pequeños circuitos cuánticos a parches locales de la imagen para generar mapas de características cuánticos, aprovechando propiedades como la superposición y el entrelazamiento. Estos mapas se procesan después con una red clásica (CNN o MLP), permitiendo explorar si los circuitos cuánticos pueden extraer patrones no triviales que complementen o mejoren las representaciones aprendidas por modelos tradicionales.

0. Imports, configuración básica y dispositivo

Para la elaboración del código es necesario importar librerías de Python, PyTorch, MedMNIST, PennyLane y sklearn. Y es importante recalcar que toda esta carpeta de códigos trabaja con las versiones y gráfica:

- Python: 3.11.14
- Torch: 2.5.1+cu121 | CUDA: True
- GPU: NVIDIA GeForce RTX 3090
- **PennyLane: 0.37.0**

Además, se ajustan 2 configuraciones importantes de PyTorch antes de comenzar el entrenamiento, ya que afectan la estabilidad numérica del modelo y el rendimiento de las convoluciones en GPU.

torch.set_default_dtype(torch.float32)

Forzar que los **tensores** flotantes que se creen de aquí en adelante usen por defecto **float32** lo hago porque PyTorch puede generar tensores en float64 (particularmente en CPU), mientras que la mayoría de las operaciones en GPU están optimizadas para float32.

Si lo quito aparecen advertencias durante el entrenamiento, con esta línea garantizo que todo el flujo de datos, pesos y gradientes sea uniforme y compatible con la GPU.

torch.backends.cudnn.benchmark = True

Cuando PyTorch usa la GPU para hacer convoluciones, tiene disponibles **varias implementaciones posibles** del mismo cálculo. Algunas son más rápidas que otras dependiendo del tamaño de la imagen, el tamaño del batch, el kernel, el stride, etc. Si no activo nada, PyTorch simplemente usa una implementación genérica. Antes de entrenar, prueba varias formas de ejecutar mis convoluciones y quédate con la más rápida para este tamaño de entrada. Luego usa siempre ese algoritmo óptimo durante todo el entrenamiento.

Esta optimización solo afecta las convoluciones clásicas de PyTorch, así que no acelera la parte de computación cuántica.

1. Hiperparámetros

En esta sección junto todos los hiperparámetros que controlan el comportamiento del experimento, tanto del modelo cuántico como de la parte clásica. Es una forma de tener toda la configuración en un solo lugar para que, si en algún momento quiero repetir el experimento o compararlo con otra versión, pueda modificar valores desde aquí sin tocar el resto del código.

En PneumoniaMNIST, la clase **1** representa *neumonía* y la clase **0** *sano*.

Los siguientes parámetros fueron escogidos para realizar la comparativa al leer todos los proyectos anteriores.

Tabla 1. Parámetros generales de entrenamiento

BATCH_SIZE	Número de imágenes que se procesan en cada batch durante una iteración.
	DataLoader(..., batch_size=BATCH_SIZE, ...)
EPOCHS	Número total de pasadas completas sobre el conjunto de entrenamiento.
	for ep in range(1, EPOCHS + 1)
LR	Tasa de aprendizaje del optimizador Adam. Para actualizar parámetros clásicos y cuánticos.
	torch.optim.Adam(...)
WEIGHT_DECAY	Factor de regularización L2 aplicado a los pesos del modelo clásico.
	weight_decay=WEIGHT_DECAY, a Adam

Tabla 2. Configuración del bloque cuántico

PATCH_SIZE	Tamaño del parche cuadrado que se mapea al circuito cuántico.
	QUBITS = PATCH_SIZE * PATCH_SIZE
QUBITS	Número total de qubits del circuito cuántico, uno por píxel del parche.
	wires=QUBITS → for w in range(QUBITS)
LAYERS	Número de capas del ansatz cuántico.
	for l in range(LAYERS)

Tabla 3. Parámetros de PennyLane

DIFF_METHOD "backprop"	Método de diferenciación que usará PennyLane para calcular gradientes.
	diff_method=DIFF_METHOD → @qml.qnode(...)
SHOTS	Número de shots (mediciones) en el simulador cuántico; None = determinista.
	qml.device(..., shots=SHOTS, ...)

En este trabajo fijo **DIFF_METHOD = "backprop"** porque uso el dispositivo **default.qubit.torch** sobre **GPU** y necesito integrar la parte cuántica dentro de un modelo PyTorch que se entrena de forma clásica. Métodos alternativos como ***parameter-shift*** o ***adjoint*** son teóricamente válidos, pero en este escenario resultan poco prácticos:

- *parameter-shift* implicaría un coste prohibitivo al aplicar la quanvolución 2×2 sobre miles de parches
- *adjoint* está más orientado a simuladores específicos sin integración directa con el flujo típico de deep learning en PyTorch.

Tabla 4. Parámetros de la cabeza clásica

DROPOUT_P	Probabilidad de apagar una neurona durante el entrenamiento en la capa densa.
	nn.Dropout(p=DROPOUT_P) → ConvHead y ClassicalCNN
OUT_DIM	Dimensión de la salida del modelo.
	nn.Linear(..., out_features=OUT_DIM) → ConvHead y ClassicalCNN en BCEWithLogitsLoss.

2. Preparación y carga de datos

El conjunto **PneumoniaMNIST** forma parte del proyecto **MedMNIST**, un banco de datos médico simplificado que contiene **radiografías de tórax** clasificadas en **dos** categorías:

- 0: Pulmón normal
- 1: Neumonía

Cada imagen tiene tamaño **28 × 28 píxeles** en **escala de grises**.

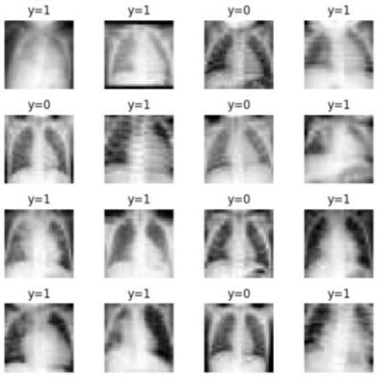


Ilustración 2. PneumoniaMNIST

2.1 Transformación básica: de imagen a tensor

Primero definimos una tranformación con **torchvision.transforms** que usando **ToTensor()** convertimos las imágenes a un **tensor** de **PyTorch**. Además, normaliza automáticamente los valores de píxel a la escala **[0, 1]** dividiendo entre 255.

2.2 Descarga y construcción de los splits de MedMNIST

Aquí construyo directamente los tres subconjuntos que ofrece MedMNIST:

```
train_set = PneumoniaMNIST(split='train', transform=tf, download=True)
val_set   = PneumoniaMNIST(split='val',   transform=tf, download=True)
test_set  = PneumoniaMNIST(split='test',  transform=tf, download=True)
```

En los tres casos:

- **transform=tf** aplica la transformación definida en 2.1 (ToTensor) a cada elemento cuando lo pido.
- **download=True** hace que, si el archivo aún no está en descargado, se descargue automáticamente.

2.3 Función auxiliar to_tensors(ds)

En esta parte **convierto los splits de PneumoniaMNIST en tensores** almacenados en CPU. Esto me asegura que todas las imágenes tengan el **mismo formato (N,1,28,28) en float32**, evita inconsistencias del dataset original y simplifica el trabajo del DataLoader, cuyos workers solo necesitan extraer slices ya preparados.

- **xs:** tensor de imágenes de forma **(N, 1, 28, 28).**
- **ys:** tensor de etiquetas de forma **(N,).**

La explicación línea por línea viene continuación:

xs = torch.stack

- **for i in range(len(ds)):** recorro todos los índices del dataset, uno por uno.
- **ds[i][0]:** tomo la **imagen** de la muestra i.
- **ds[i][0] if isinstance(ds[i][0], torch.Tensor) else torch.as_tensor(ds[i][0]):** Si ya es un tensor, lo uso tal cual; si no, lo convierto explícitamente con torch.as_tensor.
- **.to('cpu'):** fuerzo a que la imagen esté en CPU.
- **.float():** convierto el tipo a float32.
  ToTensor() ya da flotantes en [0,1], pero esta línea asegura que no haya mezclas de float64.

ys = torch.tensor

- **ds[i][1]**: tomo la **etiqueta** de la muestra i. Puede venir como tensor, numpy o escalar.
- **hasattr(ds[i][1], "item")**: si el objeto tiene método .item(), lo llamo directo.
- Si no, hago **np.asarray(ds[i][1]).item()** para asegurarme de extraer un escalar Python (0 ó 1) aunque venga como array.
- La *list comprehension* construye una lista de enteros puros [0, 1, 1, 0, ...] de longitud N.
- Luego **torch.tensor**([...], **dtype=torch.long, device='cpu')** crea un tensor entero de shape (N,) en CPU, que usaré como vector de etiquetas.

2.4 Generador de CPU y construcción de los DataLoaders

Aquí creo un generador de números aleatorios **exclusivamente en CPU**. Además, defino el número de workers que usarán los DataLoaders para cargar datos en paralelo. Usar workers acelera la lectura del dataset y dado que ya convertí todo a tensores CPU, los workers solo extraen slices.

- **TensorDataset(X_tr, y_tr)**
  Construyo un dataset simple a partir de los tensores ya preparados.
- **batch_size=BATCH_SIZE**
  Controla el tamaño de los batches.
- **shuffle=True**
  Baraja los datos en cada época.
- **generator=g_cpu**
  El shuffle se hace usando el generador de CPU con semilla, garantizando reproducibilidad.
- **num_workers=num_w**
  Cargo los datos en paralelo.
- **pin_memory=True**
  Coloca los batches en memoria "pinned", acelerando la transferencia CPU→GPU con .to(DEVICE, non_blocking=True).
- **persistent_workers=True**
  Mantiene los workers vivos entre épocas, evitando recrearlos cada vez.

2.5 Chequeo rápido de tamaños y Visualización.

Las primeras líneas son para verificar:

- Que los tres splits tienen el tamaño esperado.
- Que el train_loader entrega batches con forma (BATCH_SIZE, 1, 28, 28).

Es un sanity check que confirma que toda la pipeline de datos está correcta antes del entrenamiento.

Tabla 5. Tamaños del Dataset

Train: 4708	Val: 524	Test: 624
shape batch: torch.Size([64, 1, 28, 28])		

Como **shuffle=True**, la "**imagen 0**" de cada época no es la misma. Esto es importante porque en la visualización de la quanvolución esa imagen 0 se usará como probe, y cambiará según el shuffle.

3. Dispositivo cuántico y QNode batched

En esta sección configuro el simulador cuántico de PennyLane y defino el circuito que voy a usar como **"filtro quanvolucional 2×2"**.

La idea es que este circuito reciba muchos **parches** de imagen en **paralelo** (batched), aplique el **mismo ansatz parametrizable** a todos, y **devuelva 4 características por parche** (una por qubit).

Si bien al final del día obtuve mejores resultados en **"B_Quanvolution_CNN_PneumoniaMNIST.ipynb"** y también es el código el cuál estoy explicando, por otro lado, también he considerado que me detendré a hablar brevemente del VQA del código **"C_Quanvolution_3×3_with_lecture_in_1_qubit.ipynb"**, solo para mostrar mi idea que tuve y que no funciono como lo esperaba.

3.1 Selección del dispositivo cuántico (CPU vs GPU)

pl_torch_device = "cuda" if DEVICE.type == "cuda" else "cpu"

Como estoy entrenando mi modelo en GPU con PyTorch, quiero que el **backend** cuántico (**default.qubit.torch**) también use tensores en GPU para no estar copiando de aquí para allá. Esta sección solo me servía en el pasado para asegurarme que realmente este en GPU.

Tabla 6. Dispositivo cuántico de PennyLane.

Valor en el código	Significado técnico	Aplicación
"default.qubit.torch"	Simulador de qubits implementado sobre tensores de PyTorch. Permite que el circuito cuántico forme parte del grafo de autograd y soporte **backprop** directamente.	Es obligatorio para entrenar el VQA en GPU y permitir derivadas con PyTorch **sin recurrir a parameter-shift ni adjoint.**
wires =QUBITS = 4	Número de qubits físicos que tendrá el circuito. Cada "wire" representa un qubit independiente.	Uso **1 qubit por píxel** del parche 2×2, por lo que necesito exactamente 4 qubits para la quanvolución.
shots = SHOTS = None	Simulación analítica **sin muestreo.** Devuelve valores esperados exactos, no promedios sobre mediciones repetidas.	Evita ruido, acelera el entrenamiento y es ideal para simulación GPU. Además, permite usar backprop sin problemas.
torch_device pl_torch_device ("cuda" o "cpu")	Dispositivo donde viven **los tensores internos del simulador** (las matrices de estado, operaciones y gradientes).	Garantiza coherencia con PyTorch: si el modelo entrena en GPU, el backend cuántico también lo hace, evitando copias host–device.

Tabla 7. Configuración del QNode.

Valor	Función técnica	Aplicación en este proyecto
dev (default.qubit.torch)	Específica sobre qué **backend** cuántico vive el circuito. Es lo que respecta a la tabla de arriba	El QNode ejecuta toda la quanvolución sobre un simulador compatible con PyTorch y optimizado para GPU, manteniendo coherencia con el resto del modelo.
"torch"	Indica que el QNode debe recibir y devolver tensores de PyTorch, y enlazarse con su sistema de autograd.	Permite integrar la parte cuántica dentro del grafo computacional de PyTorch y entrenarla con optimizadores clásicos (Adam, etc.).
DIFF_METHOD = "backprop"	Define el método para calcular gradientes del circuito cuántico. "**backprop**" usa el autograd de PyTorch directamente, sin parameter-shift.	Permite derivadas eficientes en GPU, evitando overhead de parameter-shift y limitaciones del adjoint cuando hay entrada batched. Es esencial para que la quanvolución sea escalable.

Nota: En el anexo adjunto tablas breves con más información sobre la comparación de los tipos de qubits en PennyLane y el porque he optado por usar **default.qubit.torch**, además de mostrar como son optimizadas las mediciones con los diversos tipos de diferenciación.

3.2 QNode quanv_circuit_batched.

En esta parte me dedicaré a mostrar como es el QNode y que es lo que le estamos pasando, por ahora nos centraremos en el de "**B_Quanvolution_CNN_PneumoniaMNIST.ipynb".**

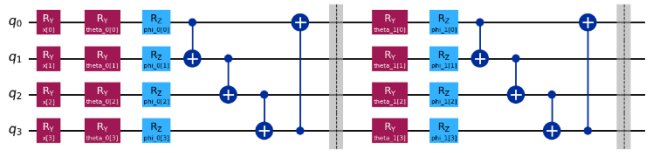


Ilustración 3. VQA: 4 qubits, feature map RY, dos capas de RY/RZ + CNOTs en anillo. La ilustración del circuito se ha realizado con Qiskit, recordemos que todo está implementado con PennyLane.

3.2.1 Entradas del circuito.

Forma de la entrada **x_angles_b.**

a) Cada imagen (28×28) se divide en **784 parches 2×2 (Son solapados)**.

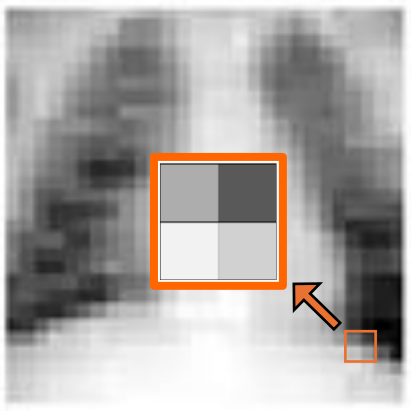


Ilustración 4. Parches 2×2.

b) Cada parche produce 4 valores, i. e. uno por píxel, uno por qubit.

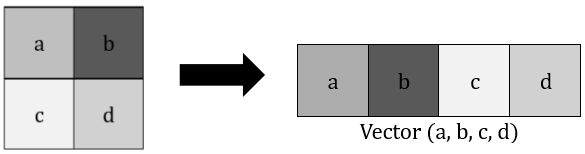


Ilustración 5. Vector de cada parche. más tarde vendrá la aclaración, pero este conjunto no son los píxeles, son ángulos en radianes dentro del intervalo [0, π]

c) Por eso la entrada al QNode tiene forma:

$$x\_angles\_b \in \mathbb{R}^{(N\_patches,\ QUBITS)}$$

Cada columna a un qubit y cada fila a un parche completo

$$x\_angles\_b = \begin{pmatrix} patch_1^{(0)} & patch_1^{(1)} & patch_1^{(2)} & patch_1^{(3)} \\ patch_2^{(0)} & patch_2^{(1)} & patch_2^{(2)} & patch_2^{(3)} \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

Forma de los parámetros entrenables **weights.**

a) Los parámetros del ansatz tienen forma:

$$weights \in \mathbb{R}^{(LAYERS,\ QUBITS,\ 2)}$$

Lo que significa:

- **LAYERS**: cuántas veces repito el bloque variacional
- **QUBITS**: uno por cada qubit
- **2 parámetros** entrenables por qubit: RY y RZ

Si quisiéramos representarlo visualmente:

weights =

Layer 0:
Q0: [θ00_RY, θ00_RZ]
Q1: [θ01_RY, θ01_RZ]
Q2: [θ02_RY, θ02_RZ]
Q3: [θ03_RY, θ03_RZ]
Layer 1:
Q0: [θ10_RY, θ10_RZ]
Q1: [θ11_RY, θ11_RZ]
Q2: [θ12_RY, θ12_RZ]
Q3: [θ13_RY, θ13_RZ]

Ilustración 6. Tensor visual de la forma de weights.

3.2.2 Codificación de datos (feature map batched)

```
for w in range(QUBITS):
    qml.RY(x_angles_b[:, w], wires=w)
```

Recorro cada qubit w = 0, 1, 2, 3. Para el qubit w, aplico una rotación RY con ángulos x_angles_b[:, w], es en esta etapa donde se coloca a los qubits en superposición con amplitudes dependientes del parche.

$$U_{\text{emb}}(\boldsymbol{\phi}) = \bigotimes_{w=0}^{3} R_Y^{(w)}(\phi_w)$$

Nota super importante: Punto clave: **x_angles_b[:, w]** no es un escalar como en los anteriores códigos, es un **vector** de tamaño N_patches. PennyLane (con default.qubit.torch) interpreta esto como un **batch de circuitos** y en todos aplica la **misma secuencia de puertas**

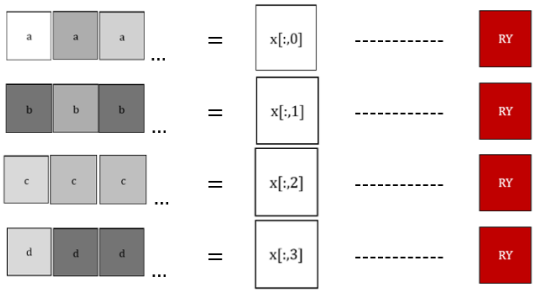


Ilustración 7. Feature map batched.

3.2.3 Ansatz variacional.

```
for l in range(LAYERS):
  for w in range(QUBITS):
    qml.RY(weights[l, w, 0], wires=w)
    qml.RZ(weights[l, w, 1], wires=w)
  qml.CNOT([0,1]); qml.CNOT([1,2]); qml.CNOT([2,3]); qml.CNOT([3,0])
```

Este es un VQA sencillo y es el que he estado ocupando en los últimos notebooks porque el objetivo hasta ahora había sido buscar que esto sea escalable y estaba más enfocado en la poder mejorar la parte clásica.

Recorro las capas variacionales $l = 0, \dots, LAYERS - 1$ y a cada capa le coloco dos bloques repetidos de rotaciones + CNOTs.

Por **cada capa** se aplican, **por qubit**, dos rotaciones entrenables: $R_Y\left(\theta_{w,y}^{(l)}\right)$ y $R_Z\left(\theta_{w,z}^{(l)}\right)$. Estos parámetros **son compartidos por todos los parches**: el mismo ansatz se aplica a cada parche, pero sobre estados iniciales distintos (porque los ángulos de entrada cambian).

$$U_{\text{rot}}^{(\ell)}(\boldsymbol{\theta}_\ell) = \bigotimes_{w=0}^{3} \left[R_Z^{(w)}(\theta_{\ell,w,z}) R_Y^{(w)}(\theta_{\ell,w,z}) \right]$$

Luego se aplica un **anillo de CNOTs** (0→1→2→3→0) que **entrelaza** a todos los qubits del parche.

$$U_{\text{ent}} = \text{CNOT}_{3\to0}\ \text{CNOT}_{2\to3}\ \text{CNOT}_{1\to2}\ \text{CNOT}_{0\to1}$$

Utilice **LAYERS=2** pues me parece que obtiene un buen compromiso entre **expresividad** y **costo**. Más capas implica más poder representacional pero más tiempo/memoria.

$$U_{\text{layer}}^{(\ell)}(\boldsymbol{\theta}_\ell) = (\text{CNOT}_{3\to0}\ \text{CNOT}_{2\to3}\ \text{CNOT}_{1\to2}\ \text{CNOT}_{0\to1}) \left(\bigotimes_{w=0}^{3} [\, R_Z^{(w)}(\theta_{\ell,w,z}) R_Y^{(w)}(\theta_{\ell,w,z})] \right)$$

$$U_{\text{layer}}^{(\ell)}(\boldsymbol{\theta}_\ell) = U_{\text{ent}}\ U_{\text{rot}}^{(\ell)}(\boldsymbol{\theta}_\ell)$$

3.2.4 Medición de los cuatro qubits

El Embedding y el Ansatz queda definido completo como:

$$U(\boldsymbol{\phi}, \boldsymbol{\theta}) = \left(\prod_{\ell=L}^{1} U_{\text{layer}}^{(\ell)}(\boldsymbol{\theta}_\ell) \right) U_{\text{emb}}(\boldsymbol{\phi})$$

Obs. El producto está ordenado: $\ell = 1$ actúa **antes** que $\ell = 2$. Pues no olvidemos que el orden va de derecha a izquierda en el producto matricial. El **estado de salida**:

$$|\psi_{\text{out}}(\boldsymbol{\phi}, \boldsymbol{\theta})\rangle = U(\boldsymbol{\phi}, \boldsymbol{\theta})\, |\psi_0\rangle$$

i. e.

$$|\psi_{out}\rangle = \left[\prod_{\ell=L}^{1}\left(\mathrm{CNOT}_{3\to 0}\mathrm{CNOT}_{2\to 3}\mathrm{CNOT}_{1\to 2}\mathrm{CNOT}_{0\to 1}\bigotimes_{w=0}^{3}R_Z^{(w)}(\theta_{\ell,w,z})R_Y^{(w)}(\theta_{\ell,w,y})\right)\right]\left(\bigotimes_{w=0}^{3}R_Y^{(w)}(\phi_w)\right)|0000\rangle$$

Para regresar a valores clásicos, se miden las **4 expectativas** $\langle Z_0\rangle, \langle Z_1\rangle, \langle Z_2\rangle, \langle Z_3\rangle \in [-1,1]$.

```
return (qml.expval(qml.PauliZ(0)),
        qml.expval(qml.PauliZ(1)),
        qml.expval(qml.PauliZ(2)),
        qml.expval(qml.PauliZ(3)))
```

Esto es el **vector de características cuánticas** del parche y más adelante veremos cómo se devuelve en cuatro canales (uno por qubit).

Ilustración 8. Flujo de los parches a través del circuito cuántico.

Algunos tipos de medición en PennyLane.

- **qml.probs()**

Medición en la base computacional, devuelve las probabilidades esperadas de cada estado base.
Salida: valores continuos entre 0 y 1 que suman 1.
Ejemplo: [0.05, 0.10, 0.25, 0.60]

- **qml.expval(qml.PauliZ(0))**

Devuelve una expectación (valor medio) del observable Pauli-Z sobre el qubit 0.
Salida: valor continuo en el rango [-1, +1].

- **qml.sample(qml.PauliZ(0))**

Realiza mediciones discretas (colapsos) del qubit 0 según los *shots* definidos.
Salida: una lista de 0 o 1 (ó ±1) por cada *shot*.
Ejemplo con shots=5: [1, -1, 1, 1, -1]

3.3 Quanvolución 3×3 con lectura en 1 qubit.

Explicación del VQA del código:

"C_Quanvolution_3×3_with_lecture_in_1_qubit.ipynb"

El filtro cuántico trabaja sobre una rejilla fija de **nueve qubits dispuestos en una vecindad 3×3**, donde el qubit central juega un papel especial.

Tabla 8. QLFO-9.

q_0	q_1	q_2
q_3	q_4	q_5
q_6	q_7	q_8

Quantum. Toda la transformación se realiza mediante un QNode y un ansatz parametrizado
Local. Se refiere a que el filtro opera sobre una **vecindad local** de la imagen.
Feature. Actúa como un **extractor de características**, como haría un filtro convolucional clásico.
Operator Hace referencia a que es un **operador cuántico** entrenable, es decir, un ansatz + un patrón de compuertas específicas (la estrella CRY).

A) Feature Map

Los nueve valores de la vecindad 3×3 se codifican en nueve qubits: RY fuerte para todos y RX débil solo para los periféricos. Esto separa desde el inicio el papel del centro y de los vecinos.

B) Ansatz variacional

Cada qubit pasa por un bloque entrenable RZ–RY–RZ, que permite al circuito ajustar y transformar la información de manera no lineal.

Los ocho qubits periféricos aplican compuertas CRY hacia el qubit central, concentrando en él toda la información del vecindario.

C) Lectura

Solo se mide el qubit central; su valor esperado en Z resume todo el procesamiento cuántico y se usa como feature para la red clásica. Que sale a un solo canal.

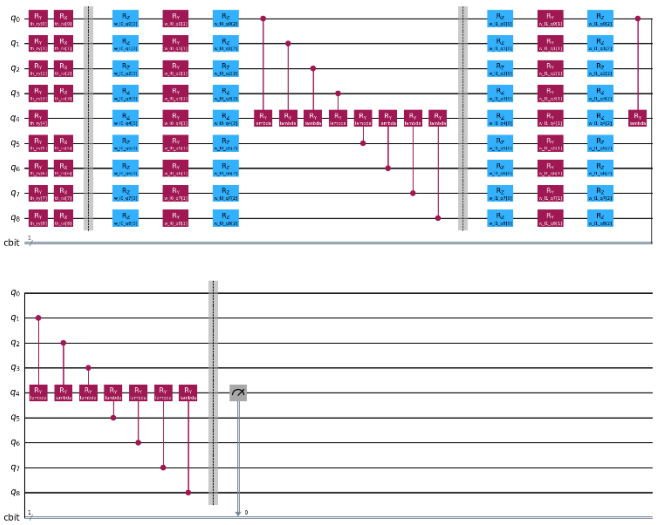


Ilustración 9. Circuito QLFO-9 utilizando Qiskit.

4. Capa Quanv2x2_28

Esta clase implementa mi "capa quanvolucional 2×2" como un nn.Module de PyTorch.

Recibe imágenes clásicas de tamaño **(B,1,28,28)** y regresa mapas de características cuánticos **(B,4,28,28)** (un canal por qubit).

La capa **Quanv3x3_QLFO9** del notebook QLFO-9, tiene algunas variaciones, pero la idea central la explicaré también se explica con **Quanv2x2_28**.

4.1 Constructor: inicialización de parámetros cuánticos

En el constructor defino los parámetros entrenables del ansatz y la configuración de cómo se procesarán los parches:

```
class Quanv2x2_28(nn.Module):
    def __init__(self, chunk_patches: int | None = 16384):
        super().__init__()
        self.weights = nn.Parameter(
            0.1 * torch.randn(LAYERS, QUBITS, 2, device=DEVICE)
        )
```

```
self._dbg_printed = False
self.chunk = chunk_patches
```

class Quanv2x2_28(nn.Module):

Defino una capa personalizada de PyTorch que se comporta como un módulo más de la red (igual que una Conv2d o una Linear), pero implementa mi quanvolución 2×2.

def __init__(self, chunk_patches: int | None = 16384):

Este es el constructor de la capa, Recibe un parámetro opcional chunk_patches, que me permite controlar cuántos parches 2×2 envío al circuito cuántico en cada bloque. Este argumento puede ser un entero para procesar los parches por partes y no saturar la VRAM o None, lo que haría que procese todos los parches de un solo jalón. En mi implementación, el valor por defecto es 16384, que resulta un tamaño de chunk equilibrado entre velocidad y uso de memoria.

super().__init__():

Llamo al constructor de nn.Module.

Esto es obligatorio para que PyTorch registre correctamente este módulo, sus parámetros entrenables y todas las funciones internas (.parameters(), .train(), .eval(), .to(device), etc.).

self.weights = nn.Parameter(0.1 * torch.randn(LAYERS, QUBITS, 2, device=DEVICE)):

Creo el tensor con los parámetros cuánticos del ansatz:

- **torch.randn(...)** genera valores gaussianos.
- Los multiplico por **0.1** para iniciar cerca de cero.
- Lo creo directamente en **DEVICE** que recordemos lo encontramos en la sección 0.
- Al envolverlo en **nn.Parameter**, le indico a PyTorch que debe ser **entrenado** por el optimizador.

self._dbg_printed = False

Esta es una bandera interna para control y me ha servido en todos los código que he hecho para garantizar que no hubo algún problema antes de esto.

Me permite imprimir el shape de la salida **solo la primera vez**, evitando spam en consola.

self.chunk = chunk_patches

Guardo el valor pasado como argumento. Luego, en el método forward, decide si procesar **todos** los parches de golpe (chunk=None) o procesarlos **por bloques** para no saturar la VRAM (cuando es un entero).

4.2 Método forward: de imagen clásica a 4 canales cuánticos

```
def forward(self, x):
    B, C, H, W = x.shape
    assert C == 1 and H == 28 and W == 28, "Esperaba (B,1,28,28)."
```

Primero se extraen las dimensiones y se asegura que la capa reciba exactamente imágenes 28×28 de un solo canal. Hay que estar atentos en esto para moverlo en caso de cambiar de data set o aplicar alguna reducción o cambio que modifique la dimensión.

```
xpad = F.pad(x, (0,1,0,1), mode="constant", value=0.0)      # (B,1,29,29)
patches = F.unfold(xpad, kernel_size=2, stride=1)           # (B,4,784)
patches = patches.transpose(1, 2).reshape(-1, 4)            # (B*784,4)
```

Se agrega **padding** pues es necesario para que la extracción de parches 2×2 con stride 1 **conserve el tamaño** espacial de la imagen. Sin **padding**, una imagen de **28×28 solo permite 27×27** posiciones para la ventana, produciendo **729 parches**. Al añadir un pixel de padding en el **borde derecho e inferior**, la imagen **se vuelve 29×29** y la ventana puede desplazarse **28 veces en cada eje, generando exactamente 784 parches**. Esto asegura que la salida de la capa quanvoluciones tenga dimensiones coherentes para reconstruirse como **4×28×28**. Propondría no mandar a cero, pero por ahora el objetivo era visualizar la reconstrucción.

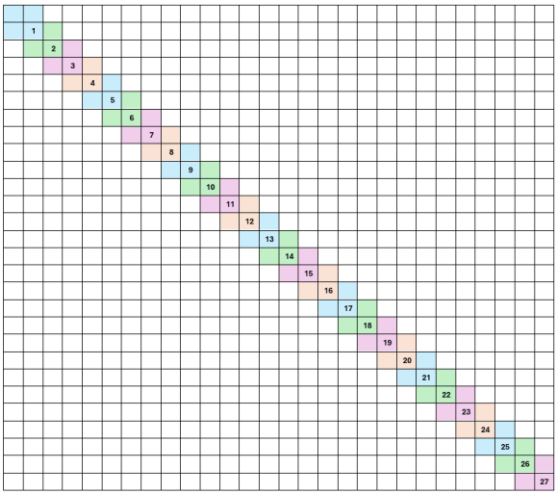


Ilustración 10. 28×28 solo permite 27×27.

Nota. Esto solo se hizo para poder realizar el rearmado, podemos modificar esta sección y dejar que se reduzca o modifique la imagen. Solo fue para poder representar el rearmado y estudiar la visualización. El ejemplo es claro, la lectura con **QLFO9** es algo diferente. Hay que tomar en cuenta que es lo que buscamos hacer.

Luego **unfold** extrae cada parche 2×2 como un **vector de 4 valores**, produciendo un tensor de **tamaño (B,4,784)**. Luego, el **transpose** reorganiza esas **784 columnas** para que cada parche sea una fila, y el **reshape(-1,4) aplana** todos los parches del batch en una sola matriz de tamaño **(B×784, 4)**. De esta manera, cada fila representa un parche listo para enviarse al circuito cuántico.

Ilustración 11. Imagen de ejemplo de cómo es la extracción de parches. La diferencia es que en el presente trabajo si se solapan.

```
angles = torch.pi * patches
```

Cada pixel del parche se convierte a un ángulo en radianes dentro del intervalo **[0, π]**, que será la entrada del conjunto de rotaciones RY del circuito cuántico.

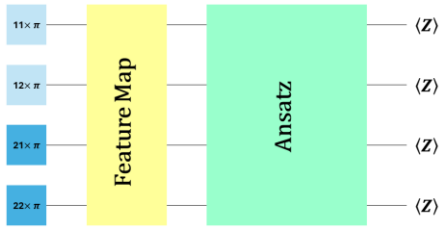


Ilustración 12. Ejemplo para aclarar como cada pixel del parche convertido a un ángulo en radianes dentro del intervalo [0, π] entra al circuito cuántico.

```
if self.chunk is None:
    m0,m1,m2,m3 = quanv_circuit_batched(angles, self.weights)
    outs = torch.stack([m0,m1,m2,m3], dim=1)
else:
    outs_list = []
    N = angles.shape[0]
    for s in range(0, N, self.chunk):
        a = angles[s:s+self.chunk]
        m0,m1,m2,m3 = quanv_circuit_batched(a, self.weights)
        outs_list.append(torch.stack([m0,m1,m2,m3], dim=1))
    outs = torch.cat(outs_list, dim=0)
```

Esta parte del código decide cuántos parches se envían al circuito cuántico en cada llamada:

a) Si **chunk=None**, todos los parches del batch (B×784) se mandan al QNode de una sola vez, produciendo cuatro vectores (uno por qubit) con las expectativas ⟨Z⟩ de cada parche.
b) Si **chunk es un entero**, entonces los parches se dividen en bloques más pequeños para evitar que la GPU se quede sin memoria; cada bloque se procesa por separado y luego todas las salidas se concatenan.

En cualquiera de los dos casos, el circuito devuelve para cada parche las **cuatro mediciones ⟨Z⟩** de los **cuatro qubits**, y estas se organizan en un **tensor (Npatches, 4)** que sigue el orden original de los parches.

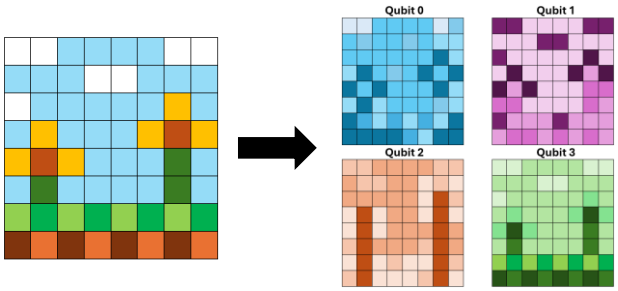


Ilustración 13. Ejemplo de los 4 canales extraídos a una única imagen. Cada qubit tiene sus pesos entrenables y ahí que cada imagen tenga algo diferente, no son pixeles ya.

```
outs = outs.view(B, 28*28, 4).permute(0,2,1).contiguous().view(B,4,28,28)
outs = outs.to(x.dtype)
```

view(B,784,4): $(B * 784,4) \rightarrow (B,784,4)$

permute(0,2,1): $(B,784,4) \rightarrow (B,4,784)$

contiguous(): $(B,784,4) \rightarrow (B,4,784)$

permute() reorganiza los ejes, pero no mueve los datos en memoria, así que el tensor queda "no contiguo" y PyTorch no puede hacer un view() sobre él. Por eso se usa contiguous(), que reescribe los datos físicamente en el nuevo orden para permitir el reshape final a (B,4,28,28) sin errores.

view(B,4,28,28): $(B,4,784) \rightarrow (B,4,28,28)$

outs = outs.to(x.dtype):

Solo ajusta el tipo de dato para que coincida con el resto del modelo.

```
if not self._dbg_printed:
    print(f"[Quanv] out: {outs.shape}, dtype={outs.dtype}, chunk={self.chunk}")
    self._dbg_printed = True
return outs
```

Este bloque imprime **una sola vez** la forma y tipo del tensor resultante para confirmar que la capa funciona correctamente. Luego marca **_dbg_printed = True para no volver a mostrar** el mensaje en los siguientes forward.

5. Cabeza convolucional (ConvHead).

Esta clase recibe los **4 canales cuánticos** de salida de la **Quanv (B,4,28,28)** y los pasa por una pequeña **CNN clásica + MLP** para producir un único logit por imagen.

5.1 Constructor: definición de la arquitectura

- **def __init__(...):** define la cabeza clásica indicando los canales de entrada (4 mapas cuánticos), la probabilidad de dropout y el número de salidas (1 logit binario).
- **super().__init__():** inicializa correctamente el módulo dentro de PyTorch para registrar parámetros y buffers.
- **self.cnn = nn.Sequential(...):** construye el bloque convolucional clásico: dos capas Conv2d con ReLU y MaxPool que reducen la imagen de 28×28 a 14×14 y luego a 7×7, generando un mapa final de características de tamaño (B,32,7,7).
- **self.dropout = nn.Dropout(...):** añade regularización apagando neuronas aleatoriamente en la parte fully-connected para evitar sobreajuste.
- **self.fc1 = nn.Linear(32*7*7, 64):** primera capa lineal que convierte el mapa aplanado en una representación intermedia de 64 dimensiones.
- **self.fc2 = nn.Linear(64, out_dim):** capa lineal final que produce el logit único por imagen (out_dim=1).

5.2 Forward: paso de los mapas cuánticos al logit final

- **z = self.cnn(z):** tomo los 4 canales cuánticos de entrada (B,4,28,28) y los paso por el bloque convolucional clásico, obteniendo un tensor compacto de características (B,32,7,7).
- **B = z.shape[0]:** guardo el tamaño del batch para poder aplanar sin perder cuántos ejemplos estoy procesando.
- **z = z.view(B, -1):** aplano cada mapa (32,7,7) en un solo vector de tamaño 32*7*7, quedando con un tensor 2D (B, 32*7*7) listo para las capas fully-connected.
- **z = F.relu(self.fc1(z)):** aplico la primera capa lineal y una activación ReLU para obtener una representación intermedia no lineal de tamaño 64.
- **z = self.dropout(z):** aplico dropout sobre esta representación para regularizar el modelo y reducir sobreajuste.
- **z = self.fc2(z):** proyecto esas 64 características a un único valor por imagen, el logit (B,1) antes de la sigmoide que se usará en la función de pérdida.

- **return z:** regreso el logit final que luego se convertirá en probabilidad de neumonía mediante sigmoide y se usará para calcular la BCEWithLogitsLoss y las métricas.

5.3 Ilustraciones de la CNN.

Tabla 9. Flujo de datos.

Bloque	Forma entrada	Operación principal	Forma salida
Conv1	(B,4,28,28)	Conv2d(4,16) + MaxPool	(B,16,14,14)
Conv2	(B,16,14,14)	Conv2d(16,32) + MaxPool	(B,32,7,7)
FC1	(B,32*7*7)	Linear + ReLU + Dropout	(B,64)
FC2	(B,64)	Linear	(B,1)

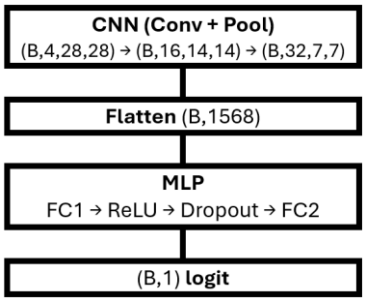


Ilustración 14. Diagrama de la red clásica.

6. Modelo completo: QCNN = Quanv2x2_28 + ConvHead.

En esta sección **ensamblo**, todo lo anterior en un solo modelo PyTorch. El flujo es:

1. **Quanv2x2_28** toma la imagen clásica y produce cuatro canales cuánticos.
2. **ConvHead** toma esos canales y produce un logit binario.

Esta clase es la que luego se entrena y evalúa. La función build_qcnn() es solo un pequeño helper para crear el modelo directamente en el dispositivo adecuado.

6.1 Definición de la clase

- **class QCNNModel(nn.Module):** defino el modelo híbrido como un módulo de PyTorch que combina la parte cuántica y la cabeza clásica.
- **def __init__(self):** constructor donde ensamblo los dos bloques principales del modelo.
- **super().__init__()** inicializa correctamente la clase base nn.Module para registrar parámetros y buffers.
- **self.quanv = Quanv2x2_28(chunk_patches=16384)** crea la capa quanvolucional 2×2 que toma (B,1,28,28) y devuelve (B,4,28,28), usando chunks de hasta 16384 parches.
- **self.head = ConvHead(in_channels=4, dropout_p=DROPOUT_P, out_dim=OUT_DIM)** define la cabeza convolucional clásica que recibe los 4 canales cuánticos y produce un logit por imagen.

6.2 Forward: encadenar Quanv + CNN

- **def forward(self, x):** define cómo fluye una imagen de entrada a través de todo el modelo.
- **zq = self.quanv(x)** aplica la capa quanvolucional: convierte la imagen clásica (B,1,28,28) en 4 mapas cuánticos (B,4,28,28).
- **out = self.head(zq)** pasa esos 4 canales por la CNN + MLP clásica para obtener un logit (B,1) por imagen.
- **return out** devuelve el logit final que luego se usará en la pérdida y en las métricas.

6.3 Helper para construir el modelo en el dispositivo

- **def build_qcnn():** función auxiliar para crear el modelo ya movido al dispositivo correcto.
- **model = QCNNModel().to(DEVICE)** instancia el modelo híbrido y lo pasa a cuda:0.
- **return model** regresa el modelo listo para entrenar.
- **model = build_qcnn()** construye efectivamente la QCNN.
- **print(model)** imprime un resumen de la arquitectura (muestra que el modelo tiene un bloque quanv y un bloque head con sus capas internas).

```
QCNNModel(
  (quanv): Quanv2x2_28()
  (head): ConvHead(
    (cnn): Sequential(
      (0): Conv2d(4, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU()
      (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): ReLU()
      (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (dropout): Dropout(p=0.2, inplace=False)
    (fc1): Linear(in_features=1568, out_features=64, bias=True)
    (fc2): Linear(in_features=64, out_features=1, bias=True)
  )
)
```

7. Función de visualización por época

Esta función sirve únicamente para inspeccionar cómo se ve la **representación cuántica** generada por la capa Quanv durante el entrenamiento.

a) Defino la función de visualización: recibe el modelo, el dataloader, el número de época y el dispositivo donde corre.
b) Pongo el modelo en modo evaluación y desactivo gradientes para evitar cómputo innecesario.
c) Tomo un batch aleatorio y lo envío al dispositivo.
d) Paso el batch solo por la capa quanvolucional, sin la CNN, para ver directamente los mapas cuánticos.
e) Selecciono la imagen 0 del batch y sus 4 canales cuánticos. También genero una versión promedio sobre los canales para visualizar una especie de "imagen cuántica global".

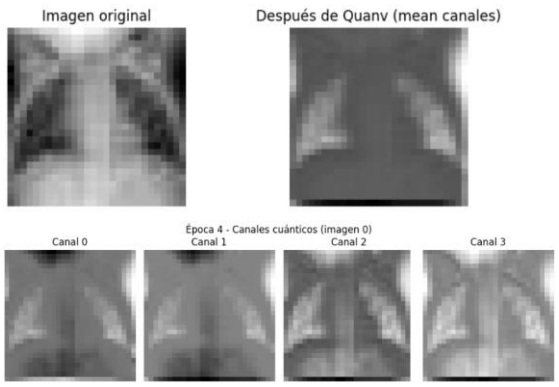


Ilustración 15. Visualización por época.

8. Entrenamiento + validación

En esta sección está todo el loop de entrenamiento y validación del modelo híbrido. Hago varias cosas, pero es tal cual como el notebook: **"PneumoniaMNIST_with_Quanvolution_2×2.ipynb"**, así que cualquier duda checar el pdf de esa carpeta donde detallo más a fondo.

8.1 pos_weight y función de pérdida

Primero contamos cuántos positivos (CLASS_POS) y negativos (CLASS_NEG) hay en el train. Luego defino:

$$\text{pos\_weight\_val} = \frac{n_{neg}}{n_{pos}}$$

para dar más peso a la clase minoritaria e imprimo el valor para tener registro del desbalance real.

Seguimos con la construcción de la perdida (**BCE con logits**), para ello usamos **nn.BCEWithLogitsLoss(...)** que espera **logits** como entrada del modelo. Podemos **aumentar el costo** de errar los positivos (clase 1) con un factor **pos_weight**.

Así para $y \in \{0,1\}$ y z el logit, la **BCE con logits** ponderada es:

$$\mathcal{L}(z,y) = (1-y)\,\log(1+e^{z}) + y(\log(1+e^{-z}) \cdot \text{pos\_w})$$

8.2 Optimizador sobre todo el modelo híbrido

```
optimizer=torch.optim.Adam(model.parameters(),lr=LR,weight_decay=WEIGHT_DECAY)
```

Ahora si **construimos el optimizador Adam**, donde:

- **model.parameters():** pasa **todos los parámetros entrenables** del modelo:
    - Los **parámetros cuánticos** (self.weights de la Quanv) que reciben gradiente vía PennyLane+Torch.
    - Los **parámetros clásicos** de la cabeza (linear y bias).
- **lr=LR:** la **tasa de aprendizaje** (float), definida en la primer sección de configuración.
- **weight_decay=WEIGHT_DECAY**: **regularización L2** sobre los pesos (ayuda a evitar sobreajuste), igualmente definidina en la primer sección de configuración.

El optimizador actualizará todos los parámetros con cada **optimizer.step()** en el lazo de entrenamiento.

8.3 Búsqueda del umbral óptimo (Youden)

Para elegir un umbral más adecuado que el fijo 0.5, implemento una función que calcula el **umbral óptimo** según el índice de Youden.

A partir de las etiquetas verdaderas y las probabilidades predichas, genero la curva ROC y obtengo los pares (fpr, tpr) para todos los thresholds posibles. Luego evalúo el índice de Youden, definido como **Youden** $=$ **TPR** $-$ **FPR**, y selecciono el umbral que maximiza esta diferencia.

Ese valor, denotado como **thr***, es el que logra el mejor compromiso global entre sensibilidad y especificidad.

8.4 Función de entrenamiento

Defino la función que entrena una época completa sobre el dataloader: **def train_epoch_q(dloader)**

La función comienza activando **model.train()**, lo que indica a PyTorch que el modelo debe comportarse en modo entrenamiento. Luego inicializa tres acumuladores: **total_loss**, **total_correct** y **total_samples**.

Enseguida entra en un loop que recorre el **dataloader**; en cada iteración recibe un batch **xb**, **yb**, lo mueve al dispositivo GPU mediante **.to(DEVICE)** y convierte las etiquetas a **float**. Después calcula los logits aplicando **model(xb)** y los aplana **con .view(-1)**. Con esos logits calcula la pérdida usando **criterion(logits, yb)**. Antes de propagar gradientes, limpia los gradientes previos con **optimizer.zero_grad()**, ejecuta **loss.backward()** para obtener los gradientes y luego actualiza todos los parámetros del modelo usando **optimizer.step()**.

Tras actualizar pesos, acumula la pérdida del batch multiplicada por su tamaño y suma al contador **total_loss**. Luego obtiene predicciones binarias comparando **logits >= 0**, porque un **logit ≥ 0** equivale a probabilidad **≥ 0.5**. Cuenta cuántas de estas predicciones coinciden con las etiquetas verdaderas y las acumula en **total_correct**. También incrementa **total_samples** con el tamaño del batch. Cuando termina el loop, calcula la pérdida promedio dividiendo **total_loss / total_samples** y la **accuracy** como **total_correct / total_samples**. Finalmente, la función regresa estas dos métricas: la pérdida promedio y la accuracy del entrenamiento durante esa época.

8.5 Función de validación

Defino la función de validación sin gradientes: **@torch.no_grad(), def eval_epoch_q(dloader):**

La función comienza con el decorador **@torch.no_grad()**, que indica que no se deben calcular gradientes. Dentro, **model.eval()** activa el modo evaluación, deshabilitando **dropout** y **batchnorm**. Luego crea dos listas vacías, **all_logits** y **all_y**, donde irá guardando todos los logits y etiquetas del set de validación. También inicializa **total_loss** y **total_samples** en cero para poder acumular la pérdida y el número total de ejemplos. Enseguida recorre cada batch del **dataloader**: mueve **xb** y **yb** al dispositivo con **.to(DEVICE)** y convierte las etiquetas a **float**. Calcula los logits pasando **xb** por el modelo y los aplana con **.view(-1)**. Con esos logits obtiene la pérdida usando **criterion(logits, yb)** y la suma a **total_loss** multiplicándola por el tamaño del batch; también acumula el total de ejemplos. Finalmente, guarda los logits y las etiquetas en sus listas, convirtiéndolos a CPU y removiendo gradientes **con .detach().cpu().**

Una vez terminado el loop, calcula la pérdida promedio dividiendo **total_loss / total_samples**. Luego concatena todos los logits acumulados con **torch.cat** y los convierte a **NumPy**, haciendo lo mismo con las etiquetas verdaderas. Convierte los logits en probabilidades aplicando la sigmoide explícitamente como **1/(1+exp(-logit))**. Después intenta calcular el **AUC ROC**; si por alguna razón falla, asigna **nan**. Llama a **_best_threshold** para obtener el umbral óptimo según el índice de **Youden**, y con este

threshold genera predicciones **y_hat_thr**, con las que calcula **accuracy** y **F1**. También genera predicciones usando el umbral estándar de 0.5 para obtener **acc_05** y **f1_05**. Finalmente, la función devuelve todas las métricas esenciales: pérdida promedio, accuracy y F1 al umbral óptimo, AUC, el propio thr_star, y accuracy/F1 al umbral 0.5.

8.6 Loop de entrenamiento QCNN

En esta parte defino el **loop principal de entrenamiento** de la QCNN. Primero creo un diccionario **history** donde voy a ir guardando, época por época, las **métricas** de **entrenamiento** (tr_loss, tr_acc05) y de **validación** (va_loss, va_auc, accuracies y F1 tanto al umbral óptimo como a 0.5, y el propio va_thr).

En **history_theta** guardo, para cada época, una copia de los **pesos cuánticos del ansatz**, para poder analizarlos después. También inicializo tres variables que llevarán el mejor modelo visto hasta ahora: **best_val_loss** arranca en infinito, **best_state** en None (aún no tengo mejor estado) y **best_thr_val** en 0.5 como valor inicial del umbral.

Luego entro al **for ep in range(1, EPOCHS + 1)**:, que recorre todas las épocas. Dentro del **loop mido el tiempo** de inicio **t0 = time.time()**, entreno una época completa con **train_epoch_q(train_loader)** y obtengo **tr_loss** y **tr_acc**.

Después evalúo en validación con **eval_epoch_q(val_loader)** y recibo todas las métricas de validación, incluyendo el umbral óptimo **thr_star**. Con eso calculo el tiempo total de la época **dt = time.time() - t0.** Si la nueva **va_loss** es menor que la mejor que tenía, actualizo **best_val_loss**, guardo una copia del estado completo del modelo en CPU como **best_state** y actualizo **best_thr_val** con el nuevo umbral óptimo.

Independientemente de si mejoró o no, agrego todas las métricas al diccionario **history** y guardo una copia de los pesos cuánticos actuales en **history_theta**. Luego imprimo un resumen formateado de la época (tiempo, pérdidas, accuracies, AUC, F1 y umbrales) y llamo a **visualize_quanv_epoch** para ver cómo está transformando la quanvolución a la imagen 0 en esa época. Al final, fuera del **loop**, si **best_state** no es **None**, restauro el mejor modelo encontrado durante el entrenamiento usando **model.load_state_dict(...)** y vuelvo a mandar los tensores al **DEVICE**, reportando por pantalla cuál fue la mejor **val_loss** y el **umbral óptimo** asociado.

```
[Info] Mejor estado QCNN restaurado (val_loss=0.0917, thr*=0.483)
```

9. Gráficas de pérdida y accuracy (Train/Val)

En esta sección solo me dedico a visualizar cómo se comportó el entrenamiento del modelo híbrido a lo largo de las épocas. Uso el diccionario **history** que llené en la sección 8 para graficar:

- La pérdida de entrenamiento y validación.
- La accuracy de entrenamiento y validación usando el umbral fijo 0.5.

Estas curvas me sirven para ver si el modelo está sobre ajustando o detectar si el aprendizaje se estanca muy pronto o si aún podría beneficiarse de más épocas. Al final también para comparar la estabilidad del entrenamiento del QCNN frente al modelo clásico CNN que defino después en la última sección.

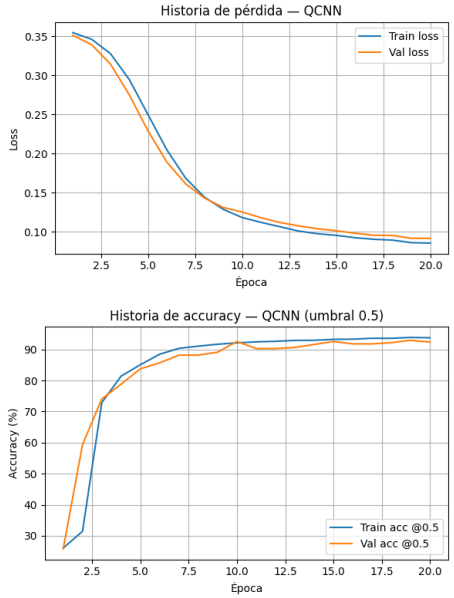


Ilustración 16. Gráficas de pérdida y accuracy (QCNN).

Las curvas muestran un entrenamiento **estable y sin sobreajuste**: la pérdida de train y validación bajan casi igual y se mantienen muy cercanas. La accuracy sube rápido y se estabiliza alrededor del **90%**, con curvas de train y val prácticamente superpuestas. En resumen, la QCNN aprende de forma consistente, converge bien y mantiene buen equilibrio entre entrenamiento y validación.

10. Test completo

Aquí ya no toco el conjunto de entrenamiento ni el de validación. Evalúo el modelo híbrido final sobre el conjunto de prueba (**test_loader**), que el modelo nunca vio durante el entrenamiento para ambos umbrales.

```
[QCNN @0.5] thr=0.500
loss=0.2824 | acc=88.6% | precision=0.894 | recall=0.928 | F1=0.911
AUC=0.923 | PR-AUC=0.912
Matriz de confusión:
 [[191  43]
 [ 28 362]]
Reporte por clase:
              precision    recall  f1-score   support

         0.0      0.872     0.816     0.843       234
         1.0      0.894     0.928     0.911       390

    accuracy                          0.886       624
   macro avg      0.883     0.872     0.877       624
weighted avg      0.886     0.886     0.885       624


[QCNN @thr*] thr=0.483
loss=0.2824 | acc=89.1% | precision=0.895 | recall=0.936 | F1=0.915
AUC=0.923 | PR-AUC=0.912
Matriz de confusión:
 [[191  43]
 [ 25 365]]
Reporte por clase:
              precision    recall  f1-score   support

         0.0      0.884     0.816     0.849       234
         1.0      0.895     0.936     0.915       390

    accuracy                          0.891       624
   macro avg      0.889     0.876     0.882       624
weighted avg      0.891     0.891     0.890       624
```

Los resultados en test muestran que la QCNN obtiene un rendimiento sólido y consistente. Con umbral fijo 0.5 alcanza **88.6% de accuracy** y **F1=0.911**, mientras que con el umbral óptimo aprendido (thr*=0.483) mejora ligeramente a **89.1% de accuracy** y **F1=0.915**. En ambos casos mantiene un **recall alto** (0.928–0.936), lo que significa que detecta la mayoría de los casos positivos. El AUC=0.923 confirma que la separación entre las

clases es buena, y la matriz de confusión muestra una reducción de falsos negativos al usar el umbral óptimo. En general, la QCNN generaliza bien y conserva un equilibrio adecuado entre precisión y sensibilidad.

11. Evolución de parámetros cuánticos (ansatz)

En esta sección tomo todos los pesos cuánticos que fui guardando en **history_theta** y los organizo para poder graficar su evolución.

Primero construyo **theta_arr** apilando, con **torch.stack**, cada copia de **model.quanv.weights** guardada en cada época: a cada tensor **(LAYERS, QUBITS, 2)** le **aplico .view(-1)** para aplanarlo, y al apilarlos obtengo un arreglo de shape **(EPOCHS, n_params)**. Luego calculo **n_params** como el número de columnas de **theta_arr** y lo imprimo para recordar cuántos parámetros totales tiene el ansatz (**LAYERS** × **QUBITS** × **2**).

Después preparo la figura donde voy a dibujar todas las curvas.

Con **plt.subplots(LAYERS, QUBITS*2, ...)** creo una rejilla de subplots con una fila por capa y dos columnas por qubit (una para RY y otra para RZ), y pongo un título general.

Luego recorro con tres índices **l** (capa), **q** (qubit) y **g** (tipo de compuerta: 0=RY, 1=RZ); en cada iteración tomo el subplot correspondiente **axs[l, 2*q + g]**, ploteo la trayectoria del parámetro **theta_arr[:, idx]** contra las épocas, le pongo título con sus índices y activo la cuadrícula.

El contador **idx** solo va avanzando para recorrer todos los parámetros. Al final ajusto el layout con **plt.tight_layout(...)**,etiqueto el eje x como "Época" y muestro la figura. Con esto obtengo, de un vistazo, cómo evoluciona **cada parámetro cuántico** del ansatz a lo largo del entrenamiento.

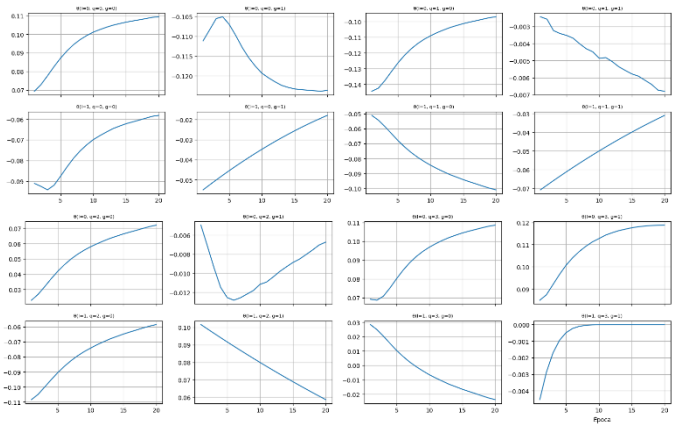


Ilustración 17. Evolución de parámetros cuánticos. Para mejor visibilidad abrir directo desde el código.

Los parámetros del ansatz cuántico que se grafican en esta sección corresponden a los ángulos de las compuertas $RY(\theta)$y $RZ(\phi)$. Aunque aparecen como valores reales sin restricciones en las gráficas, estas compuertas son **periódicas con periodo** 2π, por lo que valores como θ, $\theta + 2\pi$o $\theta - 4\pi$representan exactamente la misma operación cuántica. El optimizador (Adam) no incorpora esta periodicidad, de modo que los parámetros pueden tomar cualquier valor real durante el entrenamiento, incluso fuera del intervalo $[-\pi, \pi]$. Sin embargo, físicamente la acción de la compuerta siempre se reduce de manera efectiva a su ángulo módulo 2π.

12. Modelo clásico (CNN pura)

Finalmente, construyo una **CNN** totalmente **clásica** para tener un baseline con el que **comparar al QCNN**. La idea es:

1. Definir una arquitectura muy parecida a la cabeza clásica del modelo híbrido, pero ahora recibiendo directamente la imagen original (B,1,28,28) en lugar de los 4 canales cuánticos.

2. Entrenar esta CNN con exactamente la misma configuración de pérdida (criterion, con el mismo pos_weight), optimizador Adam, número de épocas, batch size, etc.

Guardaré de igual forma la historia de métricas igual que con el QCNN para poder comparar curvas. Evaluare en test con los dos umbrales: 0.5 y el umbral óptimo best_thr_cl que se obtiene en validación (misma lógica que antes).

```
[CLÁSICO] Mejor estado restaurado (val_loss=0.1093, thr*=0.382)
```

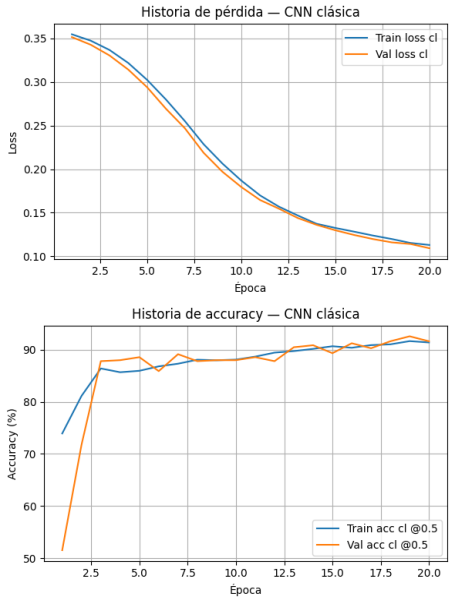
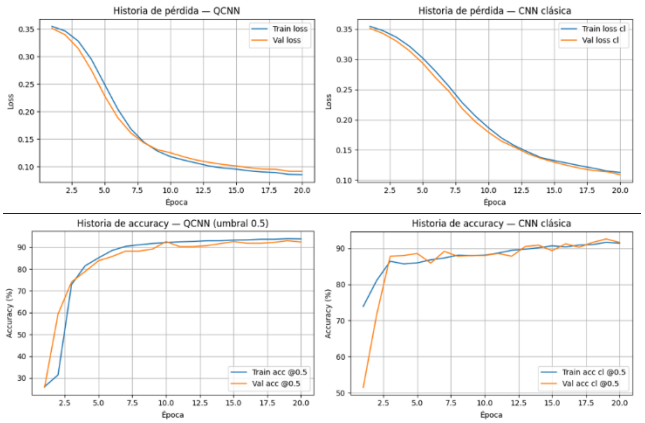


Ilustración 18. Gráficas de pérdida y accuracy (CNN).

```
[CLÁSICO @0.5] thr=0.500
loss=0.2463 | acc=87.3% | precision=0.892 | recall=0.908 | F1=0.900
AUC=0.923 | PR-AUC=0.925
Matriz de confusión:
 [[191  43]
 [ 36 354]]
Reporte por clase:
              precision    recall  f1-score   support

         0.0      0.841     0.816     0.829       234
         1.0      0.892     0.908     0.900       390

    accuracy                          0.873       624
   macro avg      0.867     0.862     0.864       624
weighted avg      0.873     0.873     0.873       624


[CLÁSICO @thr*] thr=0.382
loss=0.2463 | acc=86.2% | precision=0.858 | recall=0.933 | F1=0.894
AUC=0.923 | PR-AUC=0.925
Matriz de confusión:
 [[174  60]
 [ 26 364]]
Reporte por clase:
              precision    recall  f1-score   support

         0.0      0.870     0.744     0.802       234
         1.0      0.858     0.933     0.894       390

    accuracy                          0.862       624
   macro avg      0.864     0.838     0.848       624
weighted avg      0.863     0.862     0.860       624
```

Resultados.

I. Entrenamiento y Validación (QCNN vs CNN clásica).



Al comparar las curvas de entrenamiento y validación de ambos modelos, la QCNN y la CNN muestran comportamientos muy similares en términos de convergencia, estabilidad y tendencia general. En ambos casos, la pérdida desciende de manera monotónica, sin picos extraños y sin señales de sobreajuste marcado. Tanto la QCNN como la CNN mantienen curvas de train y validation bastante cercanas entre sí durante todo el proceso, lo que indica que ambos modelos aprenden representaciones relativamente estables para PneumoniaMNIST.

La forma de la caída del loss es prácticamente idéntica en las dos arquitecturas, y mientras la QCNN logra un val_loss ligeramente menor al final (0.0917), esta diferencia no es grande pues la CNN clásica queda un poco arriba al final (0.1093). Lo que si podemos notar es que requiere de menos épocas para lograrlo.

En cuanto a accuracy, ambas arquitecturas alcanzan valores mayores al 90% usando el umbral fijo de 0.5, y ninguna muestra una gran superioridad sólida a lo largo de todas las épocas. La QCNN crece de manera suave y sin oscilaciones grandes, mientras que la CNN clásica sube más rápido en las primeras épocas, pero presenta muchas fluctuaciones. Aun así, ambas terminan con comportamientos prácticamente equivalentes, sin que una domine total y claramente a la otra.

II. Test (QCNN vs CNN clásica)

Aunque ambas arquitecturas muestran un rendimiento sólido, la QCNN obtiene resultados consistentemente superiores al modelo clásico en las métricas más sensibles para problemas médicos: recall, F1-score y accuracy, especialmente cuando se emplea el umbral óptimo aprendido durante la validación.

El primer punto notable es que la QCNN mantiene un recall más alto que la CNN clásica en ambos umbrales. Esto significa que detecta más casos positivos de neumonía, un aspecto particularmente importante en aplicaciones clínicas donde los falsos negativos tienen un costo elevado. A umbral estándar de 0.5, la QCNN logra un recall de 0.928, mientras que la CNN clásica llega a 0.908. Con el umbral óptimo, la QCNN incluso sube a 0.936, mientras que el clásico se queda en 0.933. Aunque la diferencia no es enorme, sí es consistente y siempre a favor del modelo híbrido.

El F1-score también favorece a la QCNN en todos los escenarios. Con el umbral de 0.5, la QCNN obtiene 0.911 contra 0.900 del modelo clásico, y ajustando el umbral alcanza 0.915, su mejor desempeño. Esto confirma que el equilibrio entre precision y recall es ligeramente mejor en el modelo con capa cuántica. En términos de accuracy global, el comportamiento es similar: la QCNN se mantiene en torno a 88.6–89.1%, mientras que el modelo clásico oscila entre 86–87%.

Respecto a AUC y PR-AUC, ambos modelos empatan prácticamente en ROC-AUC con 0.923, aunque el clásico obtiene un PR-AUC apenas mayor (0.925 vs 0.912 en la QCNN). Esto indica que, en términos de ranking de probabilidades, ninguno de los dos tiene una ventaja significativa sobre el otro; son muy similares. La diferencia aparece realmente cuando se traducen estas probabilidades en decisiones binaria: allí la QCNN conserva más recalls y F1-scores.

Estos resultados muestran que, aunque la QCNN no domina completamente al modelo clásico en todas las métricas, sí lo supera de forma consistente en aquellas que importan más para la tarea de clasificación binaria en un contexto médico.

Métrica	Clásica @0.5	Clásica @thr* (0.382)
Loss	0.2463	0.2463
Accuracy	87.3 %	86.2 %
Precision	0.892	0.858
Recall	0.908	0.933
F1-score	0.900	0.894
AUC	0.923	0.923
PR-AUC	0.925	0.925
TP	354	364
FP	43	60
FN	36	26
TN	191	174

Métrica	QCNN @0.5	QCNN @thr* (0.483)
Loss	0.2824	0.2824
Accuracy	88.6 %	89.1 %
Precision	0.894	0.895
Recall	0.928	0.936
F1-score	0.911	0.915
AUC	0.923	0.923
PR-AUC	0.912	0.912
TP	362	365
FP	43	43
FN	28	25
TN	191	191

III. Canales cuánticos y evolución de parámetros.

Al observar las últimas épocas, los canales 0,1,2 y 3 presentan patrones visualmente distintos, lo cual es esperable porque cada qubit acumula una secuencia diferente de rotaciones y CNOTs. Aún no tengo certeza, pero de vista podemos ver que:

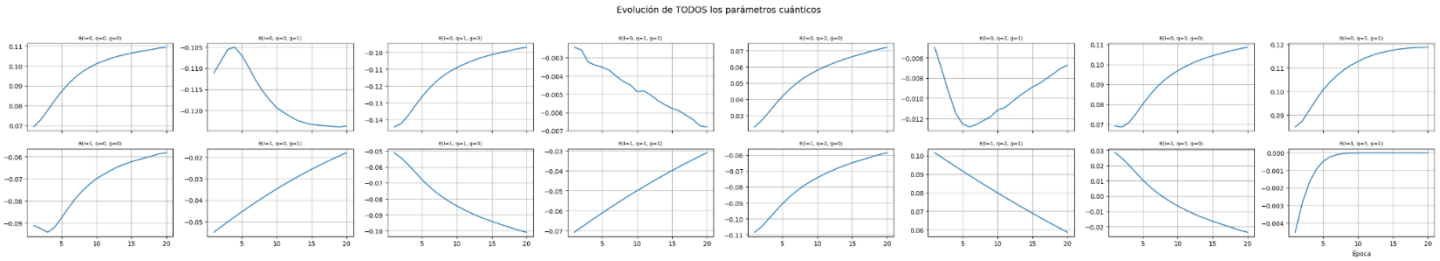
0 - Atenúa zonas oscuras, hay un buen realce de sombras en los pulmones.

1 - Representación más homogénea y conserva bordes principales y mejor definidos.

2 - Mayor contraste en región central del tórax.

3 - Tiende a saturar zonas brillantes, me parece que detecta patrones intensos

Realmente debo estudiar más esto, no era el objetivo tal cuál de este notebook, pero es importante empezar a trabajarlo desde ahora.



Al analizar los canales cuánticos a lo largo del entrenamiento, se ve claramente que el circuito no está generando ruido ni patrones aleatorios. En las primeras épocas algunos de los pesos todavía no tienen tendencia fija y tienen variación local, pero conforme avanza el entrenamiento empiezan a aparecer caminos más estables. Cada parámetro evoluciona de manera continua y sin brincos bruscos. Algunos suben, otros bajan, pero todos siguen una trayectoria bastante suave desde el inicio hasta la época 20. No hay comportamientos raros como explosiones, temblores, o cambios repentinos de dirección. Esto quiere decir que el entrenamiento sí está encontrando gradientes útiles y que el modelo no se está "atascando" ni perdiendo estabilidad.

Anexo.

Tabla 10. Comparación de dispositivos cuánticos en PennyLane.

Dispositivo	Implementación	Tipo de simulación	Gradientes compatibles	Ventajas	Limitaciones	¿Sirve para tu QCNN?
default.qubit	Numpy/Python puro	Simulación densa en CPU	**parameter-shift**, adjoint	Simple, estable, universal, buena para prototipos pequeños.	*No usa GPU*, no soporta autograd de PyTorch, lento para miles de parches.	**NO**, no puede usarse con backprop de PyTorch para entrenar la QCNN.
lightning.qubit	C++/multi-thread	Simulación densa optimizada en CPU	adjoint (muy rápido), parameter-shift	Mucho más rápido que default.qubit (5–50×), altamente optimizado.	Gradientes **NO integran PyTorch** directamente; no soporta tensores batched autograd.	**NO**, no puede usarse con backprop de PyTorch para entrenar la QCNN.
lightning.gpu	C++/CUDA (GPU)	Simulación densa optimizada en GPU	adjoint (GPU), parameter-shift	Potente para circuitos medianos, soporte avanzado en GPU.	**No** soporta autograd PyTorch; no permite entradas batched estilo torch; no funciona con backprop.	**NO** para quanvolución: no se puede entrenar vía PyTorch autograd.
default.qubit.torch	PyTorch puro (CPU/GPU)	Estado cuántico como tensor de PyTorch	**backprop nativo**, parameter-shift	*Único* que integra nativamente PyTorch, soporta GPU, soporta batch, se entrena con Adam como cualquier red neuronal.	Solo simulación densa; puede ser más lento que lightning en ciertos casos (pero escalable con GPU).	**SÍ. Es el único correcto** para QCNN + PyTorch + entrenamiento end-to-end.

La decisión de usar **"backprop"** no es arbitraria, está completamente ligada al dispositivo que uso y al hecho de que todo corre en GPU.

dev = qml.device("default.qubit.torch", wires=QUBITS, shots=SHOTS, torch_device=pl_torch_device)

Tabla 11. Comparación de métodos de diferenciación en PennyLane.

Método	Cómo calcula gradientes	Ventajas	Limitaciones	¿Funciona para tu QCNN + default.qubit.torch?
backprop	Usa el autograd nativo de PyTorch. Propaga derivadas a través del estado cuántico como si fuera cualquier tensor.	• Más rápido para circuitos pequeños/medianos. • Soporta **batching de entradas**. • Integra PyTorch de forma natural.	No funciona en dispositivos que no tengan autograd (lightning, default.qubit normal).	**SÍ y es el único método correcto para esta quanvolución.**
parameter-shift	Para cada parámetro θ evalúa el circuito dos veces: f(θ+s) y f(θ−s). Con esas mediciones estima la derivada exacta.	• Método exacto y universal. • Compatible con hardware real.	Muy lento, pues son 2 evaluaciones por parámetro. Si bien no es imposible usarlo con batches de 50,176 parches. La simulación sería muy lenta.	**NO, pues sería catastrófico por el coste 2×n_params×n_patches. Aunque sería interesante probarlo.**
adjoint	Usa el método adjunto sobre la evolución unitaria para calcular gradientes en una sola pasada eficiente.	• El método más rápido para circuitos grandes. • Gradientes exactos.	• **No soporta tensores PyTorch**. • No permite integración con QCNN híbrido. • No funciona con entradas batched.	**NO, pues no puede usarse dentro de PyTorch y tendría que ser como los notebook de las carpetas anteriores.**