# Quanvolution en PneumoniaMNIST

Universidad Nacional Autónoma de México
**Laboratorio Avanzado de Procesamiento de Imágenes**
*Académicos: Dra. Jimena Olveres Montiel y Dr. Boris Escalante Ramírez*
***Alumno: Sebastián González Juárez***

**Resumen.**

Este documento muestra la explicación de los códigos PneumoniaMNIST_with_Quanvolution_2×2, el objetivo fue tratar la idea de las Quanvoluciones a un problema con mayor cantidad de imágenes y una dimensionalidad mayor (a comparación del trabajo anterior). Para ello he propuesto el conjunto neumonía de MedMnist que cuenta con más de 5 mil imágenes con un tamaño de 28x28 píxeles.
La presente red neuronal cuántica forma parte de una colección de versiones, 6 en total, que difieren en su elección de parámetros. Entre las versiones encontramos 2 principales cambios que son trabajar con todo el conjunto de 28x28 y el otro que hace una reducción a 14x14, cada versión tiene 3 juegos de parámetros en total.

Se ha conseguido analizar los parámetros cuánticos y ha empezado a ser más escalable el proyecto, pues los códigos muestran muy buena precisión sin llegar al sobre entrenamiento.

**Palabras clave:**
Quanvoluciones, PennyLane, PyTorch, Algoritmos cuánticos variacionales, adjoint, default.qubit.

## Introducción.

### Objetivo.

Una vez que hemos aprendido a usar las **Quanvoluciones** en un conjunto pequeño como lo fue Digits Data set, vamos a buscar escalar con imágenes más grandes y ver cómo se comporta nuestro modelo. Para ello he propuesto usar el conjunto de imágenes **Pneomnoia Medmnist**. Esta red neuronal es un sistemas híbrido, para lo que ocuparemos **PyTorch** (parte clásica) y **PennyLane** (parte cuántica).

El flujo a detalle de toda la red se encuentra en la Tabla 18 en el anexo.

### Breve repaso de fundamentos de la computación cuántica.

En la computación cuántica utilizamos los **qubits** como unidad fundamental. A diferencia del bit clásico, que puede estar en 2 estados, un **qubit** puede encontrarse en una **superposición** lineal de ambas bases:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \qquad \text{con } |\alpha|^2 + |\beta|^2 = 1$$

Para llevar a cabo la **manipulación** de estos estados haremos uso de las **compuertas cuánticas**, como la Hadamard $(H)$ o las rotaciones $(R_X, R_Y, R_Z)$ y de las compuertas de control.

$$|\psi'\rangle = U|\psi\rangle, \qquad \text{con } U^\dagger U = I$$

Con el uso de estas compuertas vamos a **superposición** y **entrelazamiento** cuántico.

El proceso de medición colapsa el estado cuántico y permite obtener **valores esperados de observables** (por ejemplo, $\langle Z \rangle$), los cuales se utilizan en los modelos de aprendizaje para construir funciones de costo o vectores de características.

### Algoritmos cuánticos variacionales.

Recordemos que un VQA es un algoritmo cuántico híbrido que combina un circuito cuántico parametrizado con un optimizador clásico para minimizar la función de costo. Lo podemos dividir en 4 secciones, explicadas resumidamente a continuación:

a) **Feature Map:** Su función es codificar los datos clásicos a estados cuánticos, utilizamos compuertas cuánticas que permitan una superposición de los qubits.
b) **Variational Quantum Circuit:** Este es el núcleo del modelo cuántico, cuenta con un conjunto de compuertas dependientes de parámetros ajustables los cuales vamos a entrenar. Además, tendrá compuertas de control que tengan como objetivo entrelazar nuestros qubits. Estos dos conjuntos los podemos repetir varias veces para darle profundidad al circuito.
c) **Measurement:** Este bloque da final a la parte cuántica, en la cual medimos los estados de los qubits para obtener valores clásicos probabilistas como lo son las expectativas, estas salidas son el valor esperado de un observable y con lo que armaremos nuestra función de costo.
d) **Classical Model:** En esta sección evaluaremos la función de costo y utilizaremos un optimizador clásico para ajustar los parámetros minimizando, en esta sección se trabaja con una red neuronal clásica.

En la siguiente imagen podemos apreciarlo de mejor manera:

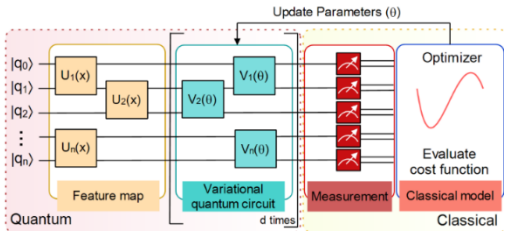



*Ilustración 1. Flujo de un algoritmo cuántico variacional.*

## Redes neuronales artificiales y convolucionales

Las **redes neuronales artificiales** son modelos computacionales inspirados en la estructura del cerebro humano, formados por unidades llamadas neuronas artificiales que **se organizan en capas**:

Una capa de entrada → Una o varias capas ocultas → Una capa de salida.

Cada neurona recibe un conjunto de **entradas ponderadas**, aplica una **función de activación no lineal** y produce una salida. El **entrenamiento consiste en ajustar los pesos** mediante un algoritmo de optimización para **minimizar una función de costo**, permitiendo que la red **aprenda patrones** a partir de los datos.

Una **CNN (Convolutional Neural Network)** aplica **filtros o kernels** sobre regiones locales de una imagen (convoluciones) para extraer características relevantes como bordes, texturas o patrones. Cada capa genera un **mapa de características** que se reduce mediante operaciones de *pooling* y finalmente se conecta a una capa totalmente conectada (*fully connected*) para la clasificación.

En el contexto de este proyecto, la red neuronal clásica posterior a la quanvolución cumple justamente esta función: recibe los mapas de activación generados por el circuito cuántico y, mediante un bloque *Flatten + MLP*, realiza la tarea de clasificación binaria entre imágenes normales y casos de neumonía.

## Quanvolutional Neural Networks (Quanv)

El modelo desarrollado en este proyecto implementa una **Quanvolutional Neural Network**, una arquitectura híbrida inspirada en las CNN clásicas, pero donde el filtro convolucional es reemplazado por un **circuito cuántico**.

El principal motivo de hacer uso de está arquitectura, fue la limitante de **no poder excedernos con el número de qubits**. Con esto podemos limitar un número de qubits y estudiar la imagen completa, esto permite trabajar con conjuntos de imágenes más grandes. En este proyecto he decidido trabajar con únicamente 4 qubits, lo cual ha sido un gran avance.

## Justificación y relevancia del enfoque híbrido

La motivación de usar un modelo híbrido radica en aprovechar el **entrelazamiento cuántico,** y las demás ventajas del cómputo cuántico, para generar características no triviales a partir de datos clásicos. Si bien actualmente el tamaño y ruido de los dispositivos cuánticos limitan su uso directo en aplicaciones a gran escala, las simulaciones como la presente permiten explorar la viabilidad de incorporar bloques cuánticos en arquitecturas convencionales.

# 1. Hiperparámetros y configuración.

En esta sección se presentan los parámetros de configuración empleados para ejecutar el modelo híbrido cuántico-clásico aplicado al conjunto de datos PneumoniaMNIST.

Se realizaron y estudiaron 6 códigos, 3 para la versión A (Sin reducción) y otros 3 para la versión B (Con reducción). En otras palabras, cada versión cuenta con sus 3 versiones dedicadas a estudiar cambios en los parámetros, los cuales están en la siguiente tabla:

*Tabla 1. Parámetros modificados para las diferentes versiones.*

| *Ipynb* | LR | Batch size | Weight decay | Épocas |
|---|---|---|---|---|
| ***A, B – 1*** | 3e-4 | 16 | 1e-3 | 8, 20 |
| ***A, B – 2*** | 3e-5 | 32 | 1e-5 | 8, 20 |
| ***A, B – 3*** | **3e-3** | **16** | **1e-4** | 8, 20 |

## 1.1. Definición del problema.

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**.

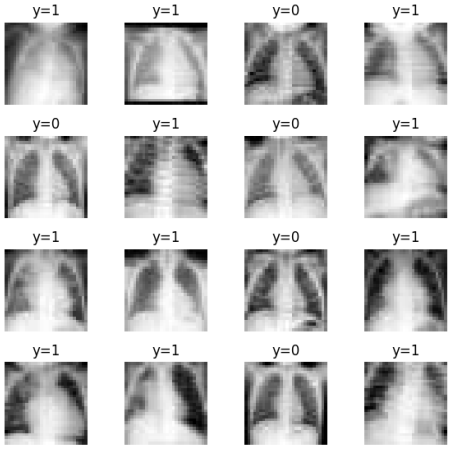



*Ilustración 2. PneumoniaMNIST.*

Si bien ya vienen etiquetadas y separadas, en el código lo volvemos a realizar. Este etiquetado proviene más que nada del trabajo anterior: **"Quanvolution2x2_Digits8x8_PyTorch_PennyLane_binario"**

Pues en ese documento buscábamos trabajar con 2 de las 10 clases disponibles, de ahí que me trajera está sección de código y permaneciera.

El hecho de que trabajemos con un conjunto de categoría binaria nos permite utilizar una función de pérdida binaria (**BCEWithLogitsLoss**) y posteriormente calcular un peso positivo (**pos_weight**) para corregir el desbalance de clases observado en el conjunto de entrenamiento.

### 1.2. Hiperparámetros de entrenamiento.

Los hiperparámetros controlan el proceso de aprendizaje del modelo clásico (MLP) y la estabilidad del entrenamiento general. Son los parámetros modificados y que difieren en las 6 versiones:

*Tabla 2. Hiperparámetros de entrenamiento*

| | |
|---|---|
| *BATCH_SIZE* | Determina el núm. de imágenes procesadas en cada iteración. |
| *EPOCHS* | Es el núm. de pasadas completas al conjunto de entrenamiento. |
| *LR* | Es la tasa de aprendizaje. |
| *WEIGHT_DECAY* | La regularización L2 aplicada a los pesos para mitigar el sobreajuste en la parte clásica. |

Para las versiones A (que trabaja sin reducción) hemos usado **8 épocas** y para las versiones B (que trabaja con reducción) hemos usado **20 épocas**. Esto fue debido al tiempo que tarda en correr el código, pues **la versión A tarda 4 veces más** y es importante destacar que **la versión B tarda alrededor de 2 horas por época** en la CPU en la que estoy trabajando. En futuros proyectos se buscará la forma de realizar la simulación con un entorno de GPU.

### 1.3. Parámetros de la Quanvolution 2×2.

En este bloque se configuran los parámetros de la quanvolución:

*Tabla 3. Parámetros de la Quanvolution 2×2.*

| | |
|---|---|
| *PATCH_SIZE = 2* | Nos indica que cada parche local es de 2×2 píxeles. |
| *QUBITS = 4* | Vamos a usar un qubit por píxel del parche. |
| *LAYERS = 2* | El circuito variacional cuántico se compone de bloques repetidos de rotaciones y entrelazamiento. |

También podríamos probar con cambiar el número de **layers**, pues esto afecta directamente en el número de parámetros entrenables. Sin embargo, en este reporte no se presentan estudios al respecto.

Se trato de utilizar el Ansatz de 16 qubits presentado en un proyecto anterior: "**Ansatz_HEA_2D_brickwork_para_VQA**".

El problema es que implicaba tener **quanvoluciones de 4x4**, lo cual en mi CPU una vez que lo corrí note que tardaría más de una semana por época. Tal vez retome esta idea una vez que tenga la posibilidad de usar la GPU.

### 1.4. Método de diferenciación y muestreo cuántico.

Para entrenar la parte cuántica se requiere un método eficiente de cálculo de gradientes. En este caso, se define:

*Tabla 4. Método de diferenciación.*

| | |
|---|---|
| *DIFF_METHOD = "adjoint"* | Activa el método adjoint differentiation de PennyLane, que calcula gradientes analíticos de forma exacta en simulación. Es más rápido que el parameter-shift y evita el ruido estadístico. |
| *SHOTS = None* | Indica un cálculo determinista de las expectativas de los observables, es decir, sin muestreo estadístico. Esto reduce la variabilidad y acelera la simulación, aunque no representa el comportamiento de un hardware real. |

### 1.5. Configuración de la red clásica (cabeza MLP).

En esta sección se define la configuración de la parte clásica del modelo, la cual procesa los mapas cuánticos obtenidos y realiza la clasificación binaria:

*Tabla 5. Configuración de la red clásica.*

| | |
|---|---|
| *USE_GAP = False* | Se desactiva el Global Average Pooling y se utiliza una capa Flatten + MLP para aprovechar toda la información espacial del mapa cuántico. |
| *DROPOUT_P = 0.20* | Apaga aleatoriamente el 20 % de las neuronas durante el entrenamiento, reduciendo el sobreajuste. |
| *OUT_DIM = 1* | Salida escalar (logit) correspondiente a la probabilidad de neumonía, antes de aplicar la función sigmoide. |

Cabe destacar que USE_GAP = False no modifica nada en el código, pues no hay salidas para True. El hecho de que no lo retirara es porque este código es una mejora directa de una de las 2 versiones que presentaba en el proyecto anterior y mencionado: **"Quanvolution2x2_Digits8x8_PyTorch_PennyLane_binario"**. En el cual se tenía una versión con USE_GAP=True y otra con False, si embargo obtuve mejores resultados para USE_GAP=False. Así que como tal cargue con esa parte sin percatarme.

## 2. Preparación y carga de datos (MedMNIST: PneumoniaMNIST)

### 2.1. Transformación y normalización.

Se define una transformación mínima para convertir cada imagen PIL/NumPy a **tensor PyTorch float32** y **escalar a** [0,1]. Dado que PneumoniaMNIST es **en escala de grises**, la forma resultante por imagen es **(1, 28, 28)**.

### 2.2. Descarga y partición del dataset.

Se descargan **los tres splits** oficiales de **MedMNIST** y se les aplica la misma transformación:

- **train**: para optimizar los parámetros del modelo.
- **val**: para selección de umbral y control de sobreajuste.
- **test**: para reporte final.

### 2.3. Conversión explícita a tensores en memoria.

Esta parte es muy **importante porque afecta directamente el rendimiento del entrenamiento** y la forma en que PyTorch accede a los datos.

Normalmente al cargar un Dataset, este no guarda las imágenes como tensores en memoria RAM, en su lugar lo que se hace es guardar rutas o arrays en disco y cada que se pide un elemento se lee esa imagen individual, la convierte en tensor, aplica las transformaciones (transform), y la devuelve. Esto ocurre cada vez que el DataLoader pide un 3atch nuevo.

Lo que hace la función es que **recorre todo el dataset una sola vez** para convertir todas la imágenes y etiquetas a tensores y **guardarlas contiguamente en memoria RAM**. De este modo nos quedan los tensores:

- **X_tr**: tensor de forma (N, 1, 28, 28)
- **Y_tr**: tensor de forma (N,)

Por ejemplo, el conjunto de entrenamiento que tiene 4788 imágenes, X_tr es un solo tensor con forma: (4788, 1, 28, 28). O suponiendo que el data set tiene 4800 imágenes y tenemos 20 épocas:

*Tabla 6. Ejemplo con 4atch_size.*

| *4atch_size* | Iteraciones por época | Total, de pasos (20 épocas) |
|---|---|---|
| *16* | 4800 / 16 = 300 | 6000 pasos |
| *32* | 150 | 3000 pasos |
| *64* | 75 | 1500 pasos |

### 2.4. DataLoaders (minilotes y barajado)

Se **crean TensorDatasets** a partir de los tensores en memoria y se configuran **tres DataLoaders**: train_loader, val_loader y test_loader.

Se crean con el tamaño de BATCH_SIZE seleccionado en la sección 1 y con **shuffle** solo en train para romper correlaciones y mejorar generalización, mientras que para val y test no se realiza para asegurar repetibilidad.

### 2.5. Verificación de tamaños.

En esta sección imprimimos los tamaños de cada uno:

**Train: 4708** | **Val: 524** | **Test: 624**

Y el shape es de la forma: torch.Size([B, 1, 28, 28]), con B= BATCH_SIZE.

## 3. Dispositivo y Qnode (adjoint, sin shots).

En esta sección **se define el bloque cuántico** que actúa como "filtro" de la **Quanvolución 2x2.** Es una algoritmo cuántico variacional y cuenta con las **3 partes** descritas en la introducción.

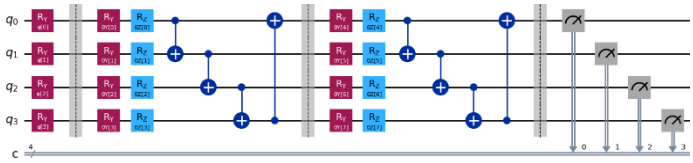


*Ilustración 3. Algoritmo cuántico Variacional utilizado, representado con Qiskit. En el podemos ver las 3 partes secciones y los 16 parámetros entrenables.*

### 3.1. Dispositivo (simulador) y modo de ejecución.

Utilizamos "**default.qubit**" como nuestro simulador vectorial, pues es rápido y precisos para pocos qubits. Los **wires** son nuestros qubits, estamos utilizando 4 justamente por que cada qubit corresponde a un pixel del parche. Recordemos que este código estamos en una simulación **determinista** y es por eso por lo que no hacemos el **muestreo**.

### 3.2. Qnode con interfaz Torch y gradiente adjoint

Con **interface="torch"** se refiere que el Qnode acepta/retorna tensores de PyTorch y es diferenciable end-to-end. Por otro lado, también necesito definir el método para el cálculo de gradientes, el cual es **diff_method="adjoint"**.

Las entradas del Qnode son 2:

- **x_angles:** ángulos de embedding (derivados del parche 2×2 ya normalizado), shape = (4,).
- **weights:** parámetros entrenables del ansatz, shape = (LAYERS, QUBITS, 2) (dos ángulos por qubit y por capa: $\theta_y, \theta_z$).

En total son **16 parámetros entrenables**, los cuales **se comparten por todos los parches**, de forma análoga a un filtro compartido en las CNN.

$$\#\theta = \text{LAYERS} \times \text{QUBITS} \times 2 = 2 \times 4 \times 2 = 16$$

### 3.3. (a) Embedding: codificación del parche en los qubits.

Se aplica $RY(\phi)$ a cada qubit, donde $\phi$ proviene del valor del píxel correspondiente. Esta etapa coloca a los qubits en superposición con amplitudes dependientes del parche (4 píxeles → 4 qubits).

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

### 3.4. (b) Ansatz variacional + entrelazamiento en anillo.

Por **cada capa** se aplican, **por qubit**, dos rotaciones entrenables: $R_Y\left(\theta_{w,y}^{(l)}\right)$ y $R_Z\left(\theta_{w,z}^{(l)}\right)$.

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

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.

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

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

### 3.5. (c) Lectura: expectativas $\langle Z \rangle$ por qubit.

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.

$$|\psi_{\text{out}}\rangle = \left[ \prod_{\ell=L}^{1} \left( \text{CNOT}_{3\to0}\text{CNOT}_{2\to3}\text{CNOT}_{1\to2}\text{CNOT}_{0\to1} \bigotimes_{w=0}^{3} R_Z^{(w)}(\theta_{\ell,w,z}) R_Y^{(w)}(\theta_{\ell,w,y}) \right) \right] \left( \bigotimes_{w=0}^{3} R_Y^{(w)}(\phi_w) \right) |0000\rangle$$

Para regresar a valores clásicos, se miden las **4 expectativas** $\langle Z_0\rangle, \langle Z_1\rangle, \langle Z_2\rangle, \langle Z_3\rangle \in [-1,1]$. Esto es el **vector de características cuánticas** del parche y se devuelve en cuatro canales (uno por qubit).

Al aplicar el Qnode **a todos los parches** y reordenar, al final vamos a construir un **mapa de activación** de tamaño (4, 14, 14) (versión A) y (4, 7, 7) (versión B).

Así la salida es una lista de **4 escalares**, así que PyTorch lo ve como un tensor (4,). Más adelante veremos como se vectoriza el reordenamiento.

Quiero hacer hincapié en que **no devuelve bits 0 o 1**, sino la expectación del observable $Z$ para cada qubit. Esta es una corrección a la ilustración 3, donde utilizo las mediciones para representar $\langle Z\rangle$.

**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]

### 3.6. Pesos compartidos por parche.

Cuando decimos que hay pesos compartidos por parche, significa que **todos los parches 2×2 de la imagen usan exactamente los mismos parámetros cuánticos (self.weights)** del circuito.

Matemáticamente: $z_i = f_{\text{QNode}}(\phi_i, \boldsymbol{\theta})$,

donde cada parche $\phi_i$ (ángulos de sus 4 píxeles) pasa por el **mismo circuito** con los **mismos parámetros entrenables** $\boldsymbol{\theta}$.

Así, no se entrena un conjunto distinto de parámetros por parche, sino **un solo circuito "filtro cuántico" que se aplica repetidamente** a todas las posiciones de la imagen.

## 4. Capas quanvolucionales 2×2 (dos variantes)

En esta sección explicare como las capas **Quanv2x2** transforman imágenes en escala de grises en **mapas de características cuánticas**.

Se cuenta con 2 versiones, pero ambas usan el **mismo circuito cuántico (QNode)** y los **mismos parámetros** para todos los parches de la imagen.

La única diferencia es el **tamaño de entrada** y, por tanto, el **número de parches** que cada imagen genera:

*Tabla 7. Características de Quanv2x2 para ambas versiones.*

| *Variante* | Entrada esperada | Núm. de parches | Tamaño del mapa resultante |
|---|---|---|---|
| *Quanv2x2_28* | (B, 1, 28, 28) | $14 \times 14 = 196$ | (B, 4, 14, 14) |
| *Quanv2x2_14* | (B, 1, 14, 14) | $7 \times 7 = 49$ | (B, 4, 7, 7) |

### 4.1. Inicialización y pesos compartidos.

En ambas versiones contamos con la misma inicialización de self.weights, que contiene los **parámetros entrenables** del circuito cuántico:

- $\theta_{\ell,w,k}$ con $\ell$=capa, $w$=qubit y $k$=rotación (RY o RZ).

Estos **pesos son los mismos para todos los parches** de todas las imágenes.

### 4.2. Entrada y comprobación.

Primero hay una sección que **garantiza que cada capa reciba el tamaño correcto** antes de extraer parches.

*Tabla 8. Verificación de entrada.*

| | Forma esperada | Verificación |
|---|---|---|
| *28×28* | (B, 1, 28, 28) | assert C==1 and H==28 and W==28 |
| *14×14* | (B, 1, 14, 14) | assert C==1 and H==14 and W==14 |

Esa sección la agregue después de haber aprendido de versiones anteriores a que es mejor poner una sección que nos indique que hemos realizado bien la dimensionalidad hasta el momento.

### 4.3. Extracción de parches 2×2.

Ambas usan **torch.nn.functional.unfold**, que en este caso nos sirve para desenrollar la imagen en bloques 2×2 sin traslape:

*Tabla 9. Número de Parches por imagen.*

| | Resultado inicial | Núm. de parches por imagen |
|---|---|---|
| *28×28* | (B, 4, 196) | 196 |
| *14×14* | (B, 4, 49) | 49 |

Después de extraerlos, el siguiente paso es reacomodar el tensor para que el parche quede como una fila independiente.

Para poderlo enviar al circuito cuántico, así que:

- transpose(1, 2) cambia el orden de los ejes, i. e. pasa de (B, 4, N) a (B, N, 4), N el núm. de parches.
- reshape(-1, QUBITS) aplana el batch, combinando todas las imágenes y parches en una sola lista.

$$\text{patches} \in \mathbb{R}^{(B\cdot N)\times 4}$$

Así que al final nos quedamos con una lista de todos los parches de Batch, cada uno con sus 4 pixeles planos.

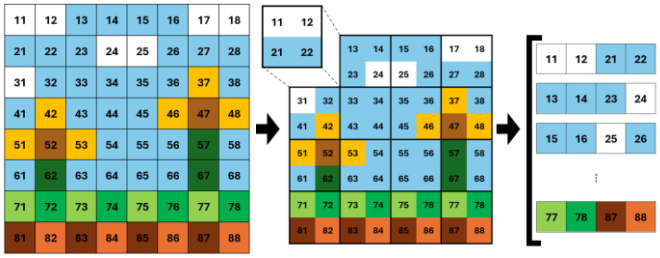


*Ilustración 4. Imagen de ejemplo de como es la extracción de parches. Se opto por no hacerlo con una imagen del conjunto, pues son muy grandes para poder hacer esta apreciación y además se hizo con color para mayor entendimiento. La imagen paso de* $8 \times 8$ pixeles a $4\times 4$ parches de 4 pixeles.

### 4.4. Aplicación del QNode y codificación de ángulos.

Cada valor se transforma en un **ángulo de rotación** para la entrada a los qubits, aprovechando que ya los habíamos normalizado:

$$\phi_i = \pi x_i$$

Así tenemos 4 ángulos que conforman al vector **x_angles** y este entra al QNode junto con los **pesos compartidos**:

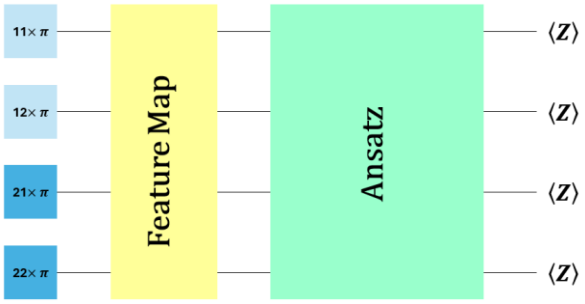



*Ilustración 5. Entrada de los 4 datos angulares al QNode. Estos datos son las intensidades en BN entre 0 y* $\pi$.

Como vimos en la sección anterior, el QNode aplica el circuito variacional con los parámetros $\boldsymbol{\theta}$: **z_i = quanv_circuit(angles[i], self.weights)**.

Y, devuelve los valores esperados $\langle Z_0\rangle, \langle Z_1\rangle, \langle Z_2\rangle, \langle Z_3\rangle \in [-1,1]$. Así que cada parche genera un vector $\mathbf{z}_i \in \mathbb{R}^4$.

Repitiendo para todos los parches:

$$\text{Total de llamadas al QNode} = \begin{cases} B \cdot 196, & \text{versión A} \\ B \cdot 49, & \text{versión B} \end{cases}$$

### 4.5. Rearmado del mapa de características.

Al haber corrido todos nuestros parches por el QNode, las salidas se fueron acumularon en la lista **outs**. De forma que **outs.shape** sería **(B * N_parches, 4)**, por ejemplo, para $B = 16$:

- En **Quanv2x2_28** → $16 \times 196 = 3136$ parches totales → outs = $(3136, 4)$
- En **Quanv2x2_14** → $16 \times 49 = 784$ parches totales → outs = $(784, 4)$

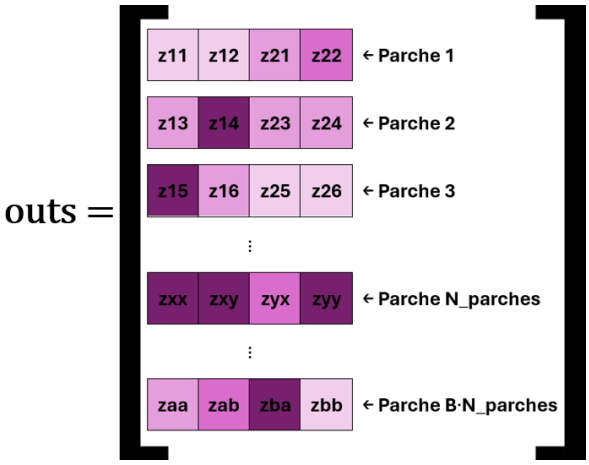



*Ilustración 6. Representación de la lista out, la cual proviene de las salidas del QNode, una lista llena de parches con 4 valores cada uno.*

Luego con **outs.reshape(B, N_parches, 4)** volvemos a separar lo que estaba plano y lo agrupamos por imagen, quedándonos una matriz por imagen con sus **N_parches** filas. I. e. cada fila es igual a 1 parche con 4 valores.

Continuando con el ejemplo:

- En **Quanv2x2_28**, outs = (3136, 4) → (16, 196, 4).
- En **Quanv2x2_14**, outs = (784 , 4) → (16, 49, 4).

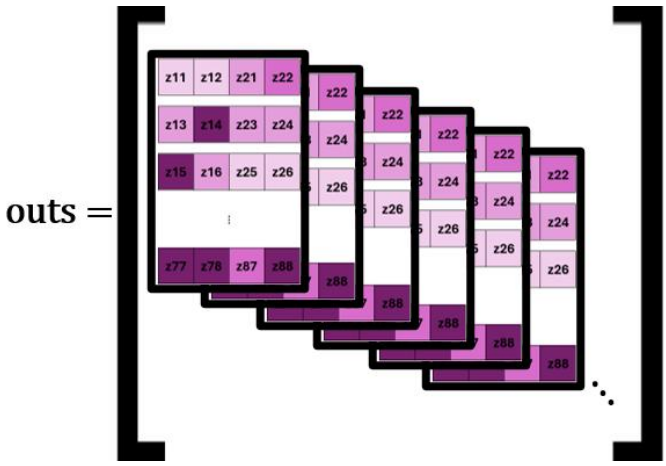



*Ilustración 7. Representación de la lista out con su rearmado de parches por imagen.*

A continuación, con **outs.permute(0, 2, 1)** cambiamos los ejes para que la dimensión donde están los 4 qubits quede como canales.

- En **Quanv2x2_28**, outs = (16, 196, 4) → (16, 4, 196).
- En **Quanv2x2_14**, outs = (16, 49, 4) → (16, 4, 49).

Lo que buscamos es formar 4 mapas (uno por qubit); por eso los 4 pasan a ser la segunda dimensión.

Para finalizar vamos a aplicar **outs.reshape(B, 4, H//2, W//2)** con lo que tomamos esos **N_parches** y los recolocamos en una grilla de $H/2 \times W/2$, recordando que los parches $2 \times 2$ eran sin translape.

- En **Quanv2x2_28**, outs = (16, 4, 196) → (16, 4, 14, 14).
- En **Quanv2x2_14**, outs = (16, 4, 49) → (16, 4, 7, 7).

De modo que la imagen original ya **no se representa como una sola matriz**, sino que se convierte en **cuatro "imágenes" diferentes** (una por qubit).

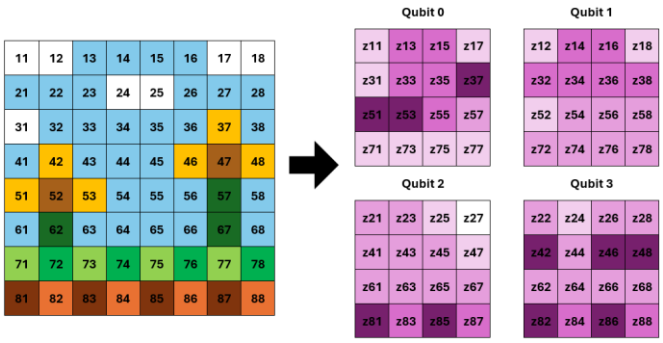



*Ilustración 8. Podemos observar la representación con la imagen de ejemplo que teníamos. Podemos ver lo que le ha pasado a una única imagen y como es que fue el rearmado, por eso tenemos 4 caneles.*

### 4.6. Consistencia y depuración

Para finalizar el **forward()**, la capa realiza dos verificaciones rápidas para asegurar que los datos se mantengan coherentes y sin errores:

(a) **Igualar tipo de dato** (outs = outs.to(x.dtype))

Convierte la salida al mismo tipo que la entrada para evitar errores por mezclar tipos (**float32 vs float64**). Garantiza que la parte cuántica y la clásica trabajen con el mismo formato numérico

(b) **Impresión de depuración** (una sola vez)

Muestra el tipo de dato de entrada y salida la primera vez que se ejecuta. Eso me permite confirmar visualmente que los tipos son correctos y que como bandera de que el error no suceda ahí, lo cual **_dbg_printed** impide que el mensaje se repita en cada batch.n

## 5. Cabezas clásicas: Flatten + MLP (2 variantes)

**IMPORTANTE:** solo ejecutar la variante que corresponda a tu salida de Quanv: 4×14×14 o 4×7×7. Pues tiene el mismo nombre y habrá problemas si compilas las dos y la que se queda es la que no utilizas.

Estas capas reciben los **mapas cuánticos** generados por la Quanv2x2 y los transforman en una **predicción final** (logit) que servirá para clasificar.

### 5.1. Inicialización.

En esta parte se **definen las capas** y sus dimensiones. Solo se ejecuta **una sola vez** lo cual es al crear el modelo con los objetos **nn.Linear** y **nn.Dropout**. Todavía no se pasan datos.

**TABLA 11. IN_FEATURES DE LA PRIMERA CAPA.**

| | Entrada. | Significado |
|---|---|---|
| **A (4×14×14)** | 784 | La Quanv salió con mapas 14×14 |
| **B (4×7×7)** | 196 | La Quanv salió con mapas 7×7 |

### 5.2. Flujo de datos del forward.

Cada vez que hacemos y = model(z), se ejecuta forward() y el dato fluye por las capas.

**TABLA 10. CABEZAS CLÁSICAS.**

| | Qué hace | Versión A | Versión B |
|---|---|---|---|
| **ENTRADA** | Mapas cuánticos de 4 canales. | (B, 4, 14, 14) | (B, 4, 7, 7) |
| **FLATTEN** | Aplana los canales en un solo vector. | (B, 784) | (B, 196) |
| **LINEAR(→64) + RELU** | Multiplica por matriz de pesos $W_1$, añade sesgo y aplica ReLU. | (B, 64) | (B, 64) |
| **DROPOUT(0.2)** | Apaga aleatoriamente 20 % de neuronas (regulariza) | (B, 64) | (B, 64) |
| **LINEAR(→1)** | Proyecta el vector a un escalar (logit). | (B, 1) | (B, 1) |

## 6. Ensamble y selección de versión del modelo

Esta sección nos va a definir el armado completo del modelo y controlará que versión utilizar (completa A o compacta B).

### 6.1. Control USE_PREPOOL

Deberemos modificar este parámetro dependiendo de la versión que buscamos y la regla es la siguiente:

- Si hay *pre-pool* a 14×14 y nos encontramos en la versión B, la cabeza debe esperar **4×7×7=196** entradas.
- Si no hay *pre-pool* nos encontramos en la versión A, la cabeza debe esperar **4×14×14=784** entradas.

Es hasta este momento donde este parámetro será usado para ver si se reducen las imágenes o no.

### 6.2. Clase contenedora para la versión compacta.

De haber escogido usar la reducción, se utilizara este modelo que encapsula el ***pre-pool***, la capa quanvolucional **14×14** y la cabeza **7×7** en un solo módulo.

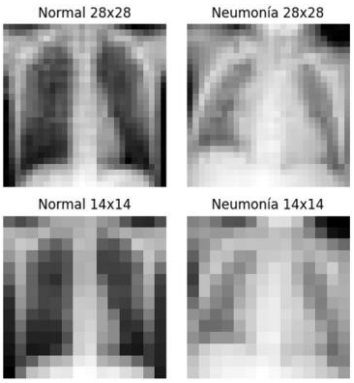



*Ilustración 9. Reducción de imagen con ejemplo de las 2 clases.*

### 6.3. Selector de Modelo.

En base a **use_prepool** se escoge el modelo de reducción o no, este ya solo hay que llamarlo y ocupar .to(DEVICE).

En caso de haber escogido la reducción, **nn.Sequential** hace la versión completa une Quanv2x2_28 con la Head(14×14).

Finalmente, solo imprimimos información del modelo elegido, como lo es la secuencia del modelo o bien el número de parámetros totales.

## 7. Entrenamiento, validación (umbral óptimo).

En esta sección veremos cómo se hizo el entrenamiento.

### 7.1. Criterio y optimizador (con pos_weight)

Primero contaremos los positivos y negativos en **train** (sin grads). Para eso usamos **with torch.no_grad()** el cual le dice a a PyTorch que **no trace gradientes** para estas operaciones (son solo contadores).

Recordemos que **y_tr** es un vector 1D con las **etiquetas de entrenamiento** (valores 0 o 1) de tamaño **N_train**. Así que sumamos en ambas clases y luego con **int(...)** convertiremos el tensor escalar a entero Python.

Continuamos con el cálculo de **pos_weight_val** para cuando hay **desbalance** (p. ej., menos neumonías que sanos), la pérdida binaria estándar tiende a favorecer a la clase mayoritaria.

En este caso **max(1, n_pos)** evita división por cero si no hubiera positivos. Lo cual no sucede pero bueno.

Seguimos con la construcción de la perdida (**BCE con logits**), para ello usamos **nn.BCEWithLogitsLoss(...)** que espera **logits** (no probabilidades) 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:

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

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.

### 7.2. Entrenamiento por época (train_epoch)

Esta función cuenta con dos parámetros, el primero es **dloader** el cual sale de nuestro **DataLoader** y en este caso nos enfocamos en **train**. Ahora usamos **model.train()** el cual activa el modo entrenamiento. Al final usamos acumuladores para promediar al final:

- **tot_loss**: suma de pérdidas *ponderadas por batch size*.
- **tot_acc**: suma de accuracies *ponderadas por batch size*.
- **tot_n**: número total de ejemplos vistos en la época.

Luego vamos a ir iterando sobre los Batches del loader, siendo:

- xb: imágenes (grises) con shape (B,1,28,28). Se mueven a DEVICE.
- yb: etiquetas {0,1} por imagen. Se mueven a DEVICE, se convierten a **float** y se aplanan a (B,).

Este shape es el que espera **BCEWithLogitsLoss** junto con logits.view(-1).

**Forward pass**: model(xb) produce **logits** (no probabilidades), shape (B,1) o (B,) según el head; el .view(-1) asegura (B,).

**Loss**: **criterion** es **BCEWithLogitsLoss** con el **pos_weight**. Compara logits vs yb y devuelve un **escalar**.

Luego al **optimizer** le metemos **zero_grad(set_to_none=True)** lo que pone a None los gradientes anteriores (más eficiente que .zero_() y evita acumulaciones).

Mediante **loss.backward()** se **retropropaga** el gradiente desde la pérdida hacia **todos los parámetros**:

- parámetros **cuánticos** (los weights de Quanv/QNode)
- parámetros **clásicos** (linear layers de la head).

Ahora con **clip_grad_norm_(..., 1.0)** hacemos **clipping** por norma L2 a 1.0 lo que significa que si la **norma L2** del vector de gradientes supera 1.0, se **escala hacia abajo** para que su magnitud total sea 1.

Finalmente, con **optimizer.step() actualizaremos** los parámetros usando Adam (usa lr y weight_decay definidos en 7.1).

Ahora tenemos el **Bloque de accuracy (acc@0.5)** el cual ponemos **logits.detach()** que corta la conexión con el grafo de gradientes para evitar que Pytorch calcule derivadas durante la métrica (solo se usa para medir, no para entrenar). Con **(logits >= 0)** convertiremos los logits en **predicciones binarias** (True/False). Lo cual es equivalente a aplicar sigmoid(logit) >= 0.5, pero es más rápido. Luego simplemente **.float()** convierte True/False a 1.0/0.0.

A continuación (preds == yb) compara predicción vs etiqueta real (0 o 1) y .mean() calcula el **promedio de aciertos** del batch, además con .item() lo convierte en un número Python (no tensor) para guardarlo o imprimirlo.

De este modo acc es la **exactitud promedio** del lote, en escala [0,1].

Continuamos con acumuladores globales, B es el tamaño del batch actual. (El último batch puede tener menos ejemplos si el total no

es múltiplo exacto del batch size.). Multiplicamos cada métrica por B para **ponderar por tamaño del batch**, así el promedio final será correcto, **por ejemplo**, no por batch.

El siguiente bloque nos servirá para ir viendo el progreso de muestra cada cierto número de iteraciones y ver como van siendo procesadas. Este me ha servido para ver que el código si esta avanzando y no ha habido algún problema.

Para terminar, se devuelven dos valores:

- tr_loss: pérdida promedio por ejemplo en la época.
- tr_acc: accuracy promedio (a umbral 0.5).

### 7.3. Validación por época y umbral óptimo (Youden J)

Primero hablemos de la función auxiliar **_best_threshold(y_true, probs)**, con está se busca calcular el **umbral óptimo** de clasificación utilizando el **índice de Youden (J)**. A partir de las etiquetas verdaderas (y_true) y las probabilidades predichas (probs), se obtiene la curva ROC, que relaciona la tasa de verdaderos positivos (tpr) con la de falsos positivos (fpr) para distintos umbrales (thr).

El índice $J = TPR - FPR$ mide qué tan bien separa el modelo ambas clases; el umbral que maximiza $J$ se considera el mejor punto de corte, thr*. Si ocurre algún error (por ejemplo, el conjunto de validación contiene solo una clase), se devuelve 0.5 por defecto. De esta manera, thr* representa el valor más equilibrado entre sensibilidad y especificidad sin depender de un criterio arbitrario.

Ahora hablemos de la función principal de este bloque, **eval_epoch(dloader)**, esta función evalúa una **época completa de validación**. Primero, el modelo se pone en modo evaluación (model.eval()) para desactivar el ***dropout*** y asegurar un comportamiento determinista. Luego recorre todo el DataLoader, pasando cada lote por el modelo para obtener sus **logits** (salidas antes del sigmoide) y calcular la pérdida con el mismo criterion usado en entrenamiento (**BCEWithLogitsLoss**). Los valores de pérdida se acumulan ponderando por el tamaño del lote, y tanto los logits como las etiquetas se almacenan para calcular métricas globales al final de la época. Todo esto se hace dentro de **torch.no_grad()** para evitar el cálculo de gradientes y reducir el consumo de memoria.

Una vez recorridos todos los lotes, los tensores de logits y etiquetas se concatenan y se transforman en probabilidades con **torch.sigmoid**. A partir de estas probabilidades, se llama a **_best_threshold** para encontrar el umbral óptimo thr* y se generan las métricas correspondientes: **accuracy**, **F1-score** y **AUC**. Además, se repiten las métricas usando el umbral estándar 0.5, lo que permite **comparar el desempeño del modelo con y sin ajuste** del umbral. El AUC (área bajo la curva ROC) se calcula de manera independiente al umbral y refleja la capacidad del modelo para separar las clases, mientras que el F1 combina precisión y *recall* en una sola medida balanceada.

La función devuelve siete valores: la pérdida promedio de validación (avg_loss), la exactitud (accS), el AUC, el F1-score (f1S), el umbral óptimo (thr_star), y las métricas estándar (acc05, f105) calculadas con umbral 0.5. En conjunto, estos resultados permiten no solo evaluar el desempeño general del modelo, sino también guardar el **umbral óptimo** para aplicarlo posteriormente en el conjunto de prueba, asegurando una evaluación justa y consistente.

### 7.4. Loop de entrenamiento

En esta sección se ejecuta el **bucle principal de entrenamiento**, que repite el proceso de *train + validación* durante todas las épocas. En cada iteración, se calculan la pérdida y exactitud de entrenamiento mediante train_epoch, y las métricas de validación con eval_epoch, incluyendo el **umbral óptimo** thr* obtenido por Youden J. Si la pérdida de validación (val_loss) mejora respecto al mejor valor previo, se guarda un **snapshot** del modelo (sus pesos) y el thr* asociado, para luego usarlo en el conjunto de prueba. Además, todas las métricas (loss, accuracy, F1, AUC y thr*) se almacenan en el diccionario history para poder graficar su evolución y analizar la convergencia del modelo. En conjunto, este bloque controla el **entrenamiento completo**, selecciona el **mejor estado del modelo** y mantiene un registro detallado del desempeño por época.

# 8. Resultados.

## 8.1. Elección de parámetros y presentación de los 6 modelos.

Se realizaron 6 modelos diferentes, con la idea de comparar los resultados de usar o no la reducción y además jugar con lo parámetros. Opte por hacer esto, pues los códigos con reducción tardan alrededor de 3 días y sin reducción casi una semana. De ser más rápido este proceso, hubiese corrido más veces para quedarme con el mejor resultado.

**Tabla 11. Configuración de los seis modelos Quanvolution**

| Modelo | Dimensión de entrada | Épocas | Batch size | Learning Rate | Weight Decay |
|---|---|---|---|---|---|
| **A1** | 28×28 | 12 | 16 | $3\times10^{-4}$ | $1\times10^{-3}$ |
| **A2** | 28×28 | 12 | 32 | $3\times10^{-5}$ | $1\times10^{-5}$ |
| **A3** | 28×28 | 12 | 16 | $3\times10^{-3}$ | $1\times10^{-4}$ |
| **B1** | 14×14 | 20 | 16 | $3\times10^{-4}$ | $1\times10^{-3}$ |
| **B2** | 14×14 | 20 | 32 | $3\times10^{-5}$ | $1\times10^{-5}$ |
| **B3** | 14×14 | 20 | 16 | $3\times10^{-3}$ | $1\times10^{-4}$ |

### A) Dimensión de entrada.

El **cambio principal** entre versiones A y B fue **reducir la resolución** de entrada a la mitad. Esto buscando reducir el costo cuántico y clásico, pues menos patches son menos qubits simulados. Además de ver si podíamos hacer esto para acelerar el proceso sin perder precisión.

### B) Número de épocas.

El incremento de épocas en las versiones **B** buscó **compensar la menor resolución** permitiendo un aprendizaje más prolongado. Sin embargo, aunque se mejorara la convergencia, no hubo un aumento muy significativo en la precisión. Considero que pudieron haber estado más épocas ambas versiones sin llegar al sobre entrenamiento, pero el tiempo me lo impedía.

### C) Batch size. (16 vs 32)

Los modelos **A2 y B2** duplicaron el tamaño de batch para probar si un mayor promedio estadístico por iteración suavizaba el gradiente. Se espera menor oscilación y más generalización, aunque el aprendizaje sea más lento.

### D) Learning Rate.

Cada trío de versiones explora **una magnitud distinta de LR**, abarcando tres órdenes de magnitud:

- **A1 y B1 ($3\times10^{-4}$):** tasa intermedia, entrenamiento estable.
- **A2 y B2 ($3\times10^{-5}$):** tasa pequeña, convergencia lenta pero segura.
- **A3 y B3 ($3\times10^{-3}$):** tasa alta, aprendizaje rápido, pero riesgo de sobreajuste.

### E) Weight Decay.

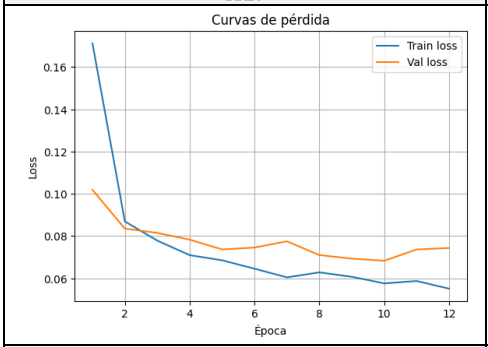
Este parámetro regula la penalización por pesos grandes (control de sobreajuste).

- **Alto ($1e^{-3}$):** regulariza fuertemente, menor sobreajuste, pero menor flexibilidad.
- **Bajo ($1e^{-5}$):** deja aprender libremente, riesgo de sobreajuste.
- **Intermedio ($1e^{-4}$):** Equilibrado.

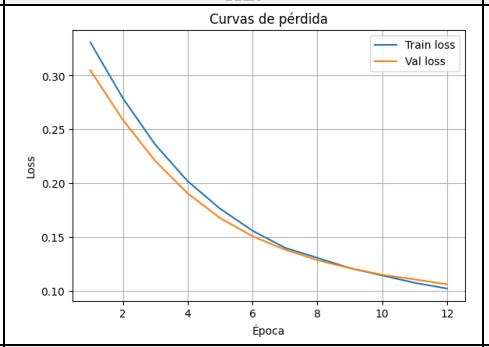
### 8.2. Curvas de pérdida (Train / Val).

**Tabla 12. Curvas de pérdida**

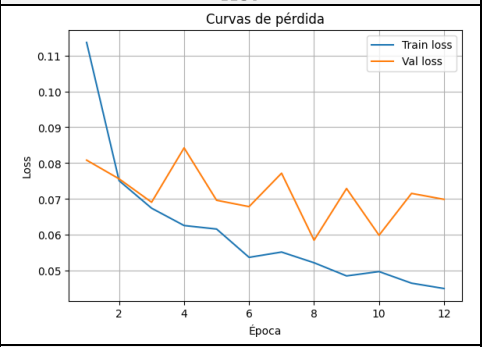
**A1.**



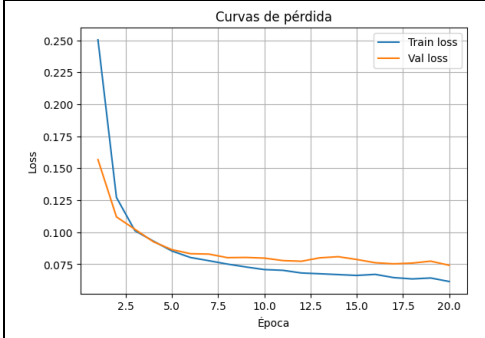

**A2.**



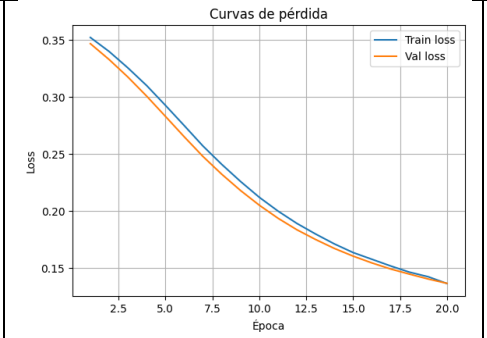

**A3.**



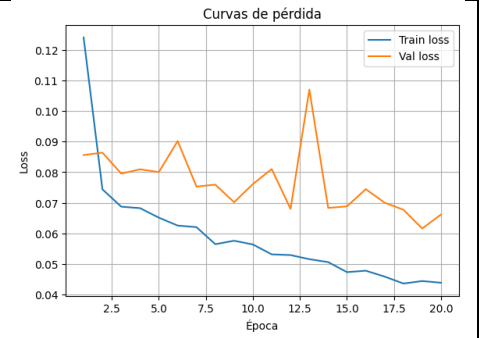

**B1.**




**B2.**




**B3.**




| Modelo | Tendencia | Sobreajuste | Observaciones |
|---|---|---|---|
| A1 | Descenso rápido inicial, luego estable. | Mínimo. | Train y Val casi paralelas, brecha <0.01 al final. |
| A2 | Convergencia lenta, estable. | No evidente. | Reducción progresiva sin oscilaciones. |
| A3 | Fluctuaciones leves en validación. | Ligero sobreajuste. | Pérdida de validación más variable, pero converge bien. |
| B1 | Descenso continuo y paralelo. | No. | Curvas muy uniformes, sin divergencia. |
| B2 | Decrecimiento suave y sostenido. | No. | Excelente estabilidad, sin brecha significativa. |
| B3 | Picos ocasionales en validación. | Parcial. | Train desciende estable; Val con oscilaciones leves. |

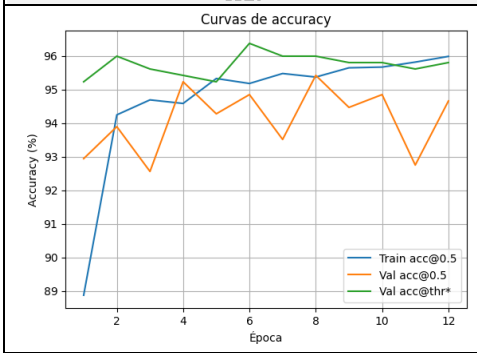
**Conclusiones de resultados:**

- Las curvas de pérdida evidencian que **todas las configuraciones lograron convergencia sin divergencias críticas**.
- Los modelos con **LR=3e−3 (A3 y B3)** presentan pequeñas oscilaciones propias del aprendizaje agresivo, pero no derivan en sobre entrenamiento marcado.
- Los modelos **A2 y B2** (LR=3e−5) fueron los más estables, aunque con ritmo de convergencia más lento.
- Reducir la imagen a **14×14 (B)** no comprometió la estabilidad.

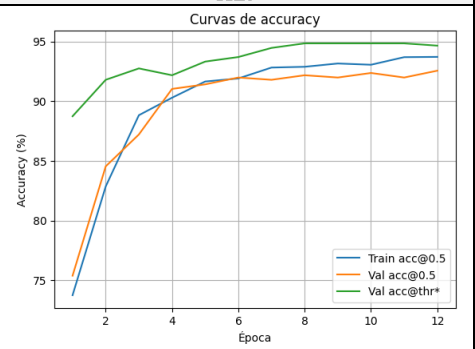
### 8.3. Curvas de exactitud (Accuracy)

**Tabla 13. Curvas de exactitud**

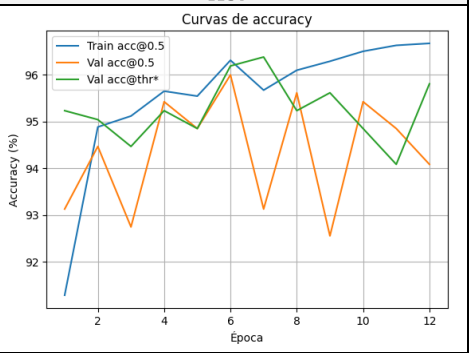
**A1.**



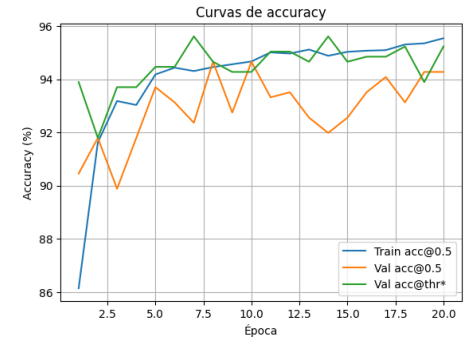

**A2.**



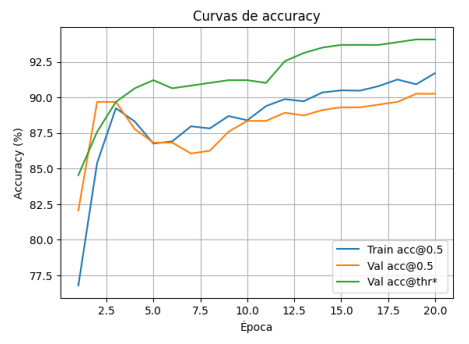

**A3.**



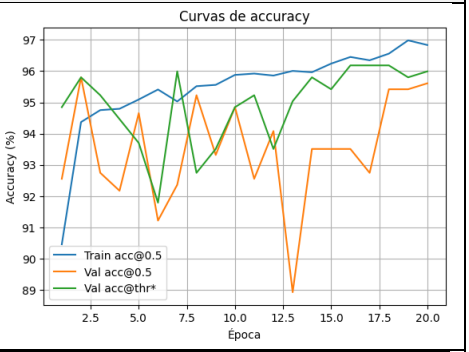

**B1.**




**B2.**




**B3.**




| Modelo | Acc. máx. Train | Acc. máx. Val (thr=0.5) | Acc. máx. Val (thr*) | Observaciones |
|---|---|---|---|---|
| A1 | 96.0% | 95.4% | 96.4% | Curvas convergentes, alta estabilidad. |
| A2 | 93.7% | 92.6% | 94.7% | Crecimiento progresivo y coherente. |
| A3 | 96.7% | 96.0% | 96.4% | Ligero sobreajuste tras época 10. |
| B1 | 95.5% | 94.3% | 95.2% | Altamente paralelo entre train y val. |
| B2 | 91.7% | 90.3% | 94.1% | Incremento sostenido, sin saltos. |
| B3 | 96.8% | 95.6% | 96.2% | Mejores valores globales, pero con claros saltos de baja estabilidad. |

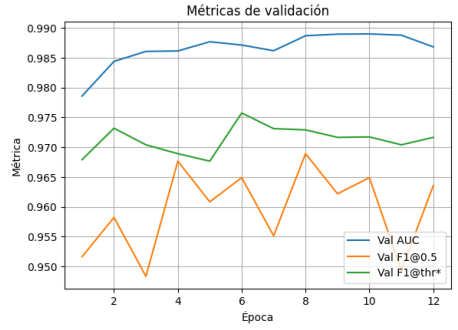
**Conclusiones de resultados:**

- Los **accuracies de validación siguen de cerca a las de entrenamiento**, lo cual indica una correcta generalización.
- El **mayor valor alcanzado (96.4%)** se observa tanto en versiones 1 como en 3, mientras que **B3** logra el valor más alto, sin embargo, con muchas ocilasiones.
- El umbral adaptativo (*thr* de Youden J) mejora marginalmente la precisión en los seis modelos, lo que confirma que **el ajuste dinámico del punto de decisión favorece la sensibilidad sin sacrificar especificidad**.

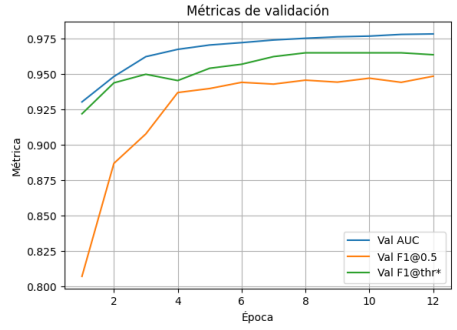
## 8.4. Métricas de validación (AUC y F1).

**Tabla 14. Métricas de validación (AUC y F1)**

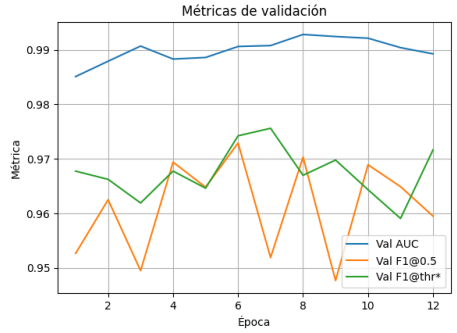
**A1.**



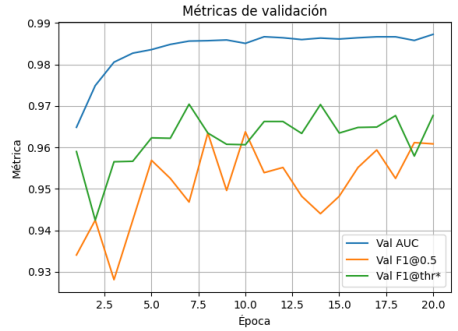

**A2.**



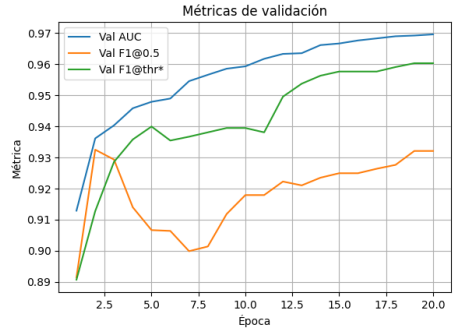

**A3.**



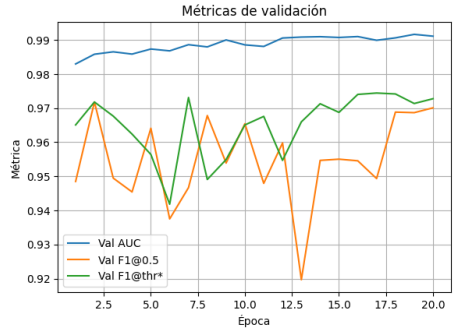

**B1.**




**B2.**




**B3.**

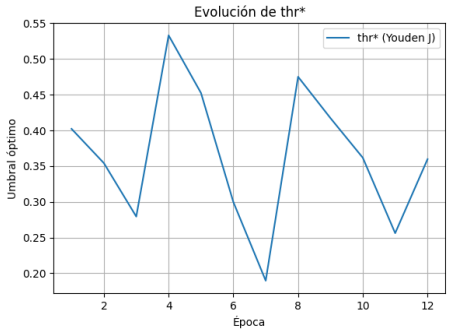



| Modelo | AUC máx. | F1 @0.5 | F1 @thr* | Observaciones |
|---|---|---|---|---|
| A1 | 0.989 | 0.965 | 0.972 | AUC alto y constante. |
| A2 | 0.979 | 0.949 | 0.964 | Estabilidad con leve rezago en F1. |
| A3 | 0.993 | 0.970 | 0.974 | Mejor desempeño general. |
| B1 | 0.987 | 0.961 | 0.968 | Consistente, sin oscilaciones. |
| B2 | 0.970 | 0.932 | 0.960 | Evolución ascendente, curva limpia. |
| B3 | 0.991 | 0.969 | 0.974 | AUC estable y cercano a A3. |

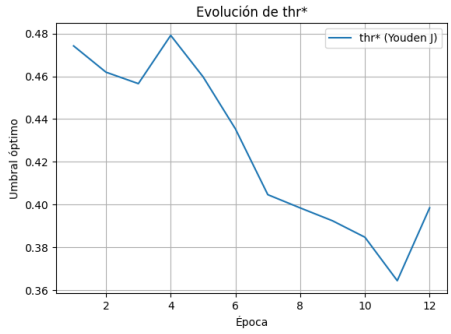
### 8.5. Evolución del umbral óptimo (thr*)

**Tabla 15. Métricas de validación (AUC y F1)**

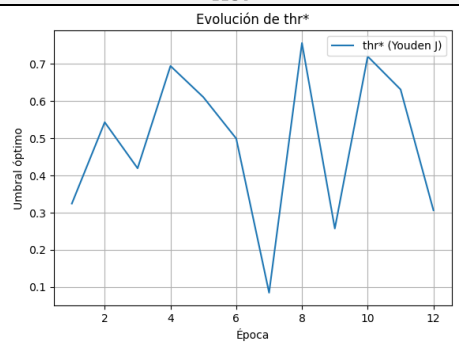
**A1.**



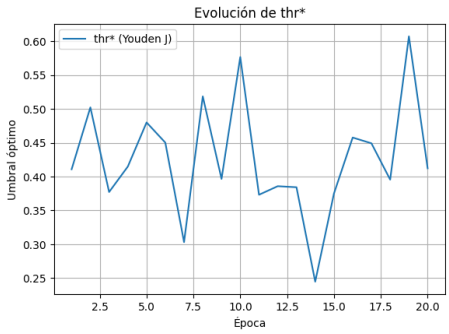

**A2.**



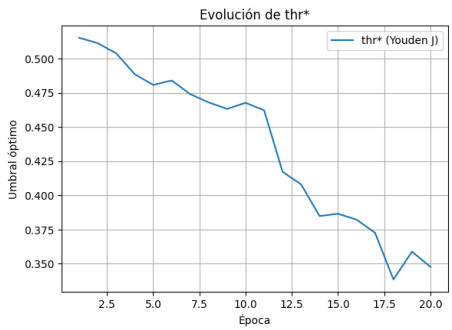

**A3.**



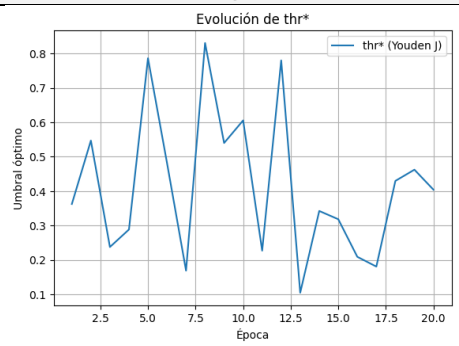

**B1.**




**B2.**




**B3.**




| Modelo | thr* inicial | thr* final | Tendencia | Observaciones |
|---|---|---|---|---|
| A1 | 0.40 | 0.36 | Fluctuante moderado | Ajuste sensible entre 0.2–0.5 |
| A2 | 0.47 | 0.38 | Descenso suave | Muestra convergencia estable. |
| A3 | 0.32 | 0.30 | Oscilante | Picos por LR alto. |
| B1 | 0.41 | 0.41 | Estable | Poca variación, confiable. |
| B2 | 0.52 | 0.34 | Descendente | Ajuste progresivo correcto. |
| B3 | 0.36 | 0.40 | Muy variable | Alta sensibilidad al ruido. |

### 8.6. Evaluación en conjunto de prueba (TEST)

**Tabla 16. Resumen de Test A.**

**A1.**

**[TEST @0.5] thr=0.500**

loss=0.3789 | acc=88.3% | precision=0.859 | recall=0.972 | F1=0.912

AUC=0.930 | PR-AUC=0.916

Matriz de confusión:

```
 [[172  62]
 [ 11 379]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.94 | 0.74 | 0.82 | 234 |
| Neumonía(1) | 0.86 | 0.97 | 0.91 | 390 |
| accuracy | | | 0.88 | 624 |
| macro avg | 0.90 | 0.85 | 0.87 | 624 |
| weighted avg | 0.89 | 0.88 | 0.88 | 624 |

**[TEST @thr*] thr=0.362**

loss=0.3789 | acc=86.9% | precision=0.838 | recall=0.979 | F1=0.903

AUC=0.930 | PR-AUC=0.916

Matriz de confusión:

```
 [[160  74]
 [  8 382]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.95 | 0.68 | 0.80 | 234 |
| Neumonía(1) | 0.84 | 0.98 | 0.90 | 390 |
| accuracy | | | 0.87 | 624 |
| macro avg | 0.90 | 0.83 | 0.85 | 624 |
| weighted avg | 0.88 | 0.87 | 0.86 | 624 |

**A2.**

**[TEST @0.5] thr=0.500**

loss=0.2821 | acc=86.5% | precision=0.866 | recall=0.928 | F1=0.896

AUC=0.915 | PR-AUC=0.908

Matriz de confusión:

```
 [[178  56]
 [ 28 362]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.86 | 0.76 | 0.81 | 234 |
| Neumonía(1) | 0.87 | 0.93 | 0.90 | 390 |
| accuracy | | | 0.87 | 624 |
| macro avg | 0.87 | 0.84 | 0.85 | 624 |
| weighted avg | 0.87 | 0.87 | 0.86 | 624 |

**TEST @thr*] thr=0.399**

loss=0.2821 | acc=85.9% | precision=0.848 | recall=0.944 | F1=0.893

AUC=0.915 | PR-AUC=0.908

Matriz de confusión:

```
 [[168  66]
 [ 22 368]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.88 | 0.72 | 0.79 | 234 |
| Neumonía(1) | 0.85 | 0.94 | 0.89 | 390 |
| accuracy | | | 0.86 | 624 |
| macro avg | 0.87 | 0.83 | 0.84 | 624 |
| weighted avg | 0.86 | 0.86 | 0.86 | 624 |

**A3.**

**[TEST @0.5] thr=0.500**

loss=0.4076 | acc=88.3% | precision=0.859 | recall=0.972 | F1=0.912

AUC=0.944 | PR-AUC=0.940

Matriz de confusión:

```
 [[172  62]
 [ 11 379]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.94 | 0.74 | 0.82 | 234 |
| Neumonía(1) | 0.86 | 0.97 | 0.91 | 390 |
| accuracy | | | 0.88 | 624 |
| macro avg | 0.90 | 0.85 | 0.87 | 624 |
| weighted avg | 0.89 | 0.88 | 0.88 | 624 |

**[TEST @thr*] thr=0.756**

loss=0.4076 | acc=90.2% | precision=0.898 | recall=0.951 | F1=0.924

AUC=0.944 | PR-AUC=0.940

Matriz de confusión:

```
 [[192  42]
 [ 19 371]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.91 | 0.82 | 0.86 | 234 |
| Neumonía(1) | 0.90 | 0.95 | 0.92 | 390 |
| accuracy | | | 0.90 | 624 |
| macro avg | 0.90 | 0.89 | 0.89 | 624 |
| weighted avg | 0.90 | 0.90 | 0.90 | 624 |

**Tabla 17. Resumen de Test B.**

**B1.**

**[TEST @0.5] thr=0.500**

loss=0.2948 | acc=89.3% | precision=0.876 | recall=0.964 | F1=0.918

AUC=0.935 | PR-AUC=0.928

Matriz de confusión:

```
 [[181  53]
 [ 14 376]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.93 | 0.77 | 0.84 | 234 |
| Neumonía(1) | 0.88 | 0.96 | 0.92 | 390 |
| accuracy | | | 0.89 | 624 |
| macro avg | 0.90 | 0.87 | 0.88 | 624 |
| weighted avg | 0.90 | 0.89 | 0.89 | 624 |

[TEST @thr*] thr=0.412

loss=0.2948 | acc=88.6% | precision=0.862 | recall=0.974 | F1=0.915

AUC=0.935 | PR-AUC=0.928

Matriz de confusión:

```
 [[173  61]
 [ 10 380]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.95 | 0.74 | 0.83 | 234 |
| Neumonía(1) | 0.86 | 0.97 | 0.91 | 390 |
| accuracy | | | 0.89 | 624 |
| macro avg | 0.90 | 0.86 | 0.87 | 624 |
| weighted avg | 0.89 | 0.89 | 0.88 | 624 |

**B2.**

**[TEST @0.5] thr=0.500**

loss=0.2431 | acc=87.8% | precision=0.903 | recall=0.903 | F1=0.903

AUC=0.920 | PR-AUC=0.912

Matriz de confusión:

```
 [[196  38]
 [ 38 352]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.84 | 0.84 | 0.84 | 234 |
| Neumonía(1) | 0.90 | 0.90 | 0.90 | 390 |
| accuracy | | | 0.88 | 624 |
| macro avg | 0.87 | 0.87 | 0.87 | 624 |
| weighted avg | 0.88 | 0.88 | 0.88 | 624 |

**[TEST @thr*] thr=0.348**

loss=0.2431 | acc=86.7% | precision=0.850 | recall=0.956 | F1=0.900

AUC=0.920 | PR-AUC=0.912

Matriz de confusión:

```
 [[168  66]
 [ 17 373]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.91 | 0.72 | 0.80 | 234 |
| Neumonía(1) | 0.85 | 0.96 | 0.90 | 390 |
| accuracy | | | 0.87 | 624 |
| macro avg | 0.88 | 0.84 | 0.85 | 624 |
| weighted avg | 0.87 | 0.87 | 0.86 | 624 |

**B3.**

**[TEST @0.5] thr=0.500**

loss=0.3405 | acc=89.4% | precision=0.880 | recall=0.962 | F1=0.919

AUC=0.943 | PR-AUC=0.953

Matriz de confusión:

```
 [[183  51]
 [ 15 375]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.92 | 0.78 | 0.85 | 234 |
| Neumonía(1) | 0.88 | 0.96 | 0.92 | 390 |
| accuracy | | | 0.89 | 624 |
| macro avg | 0.90 | 0.87 | 0.88 | 624 |
| weighted avg | 0.90 | 0.89 | 0.89 | 624 |

**[TEST @thr*] thr=0.462**

loss=0.3405 | acc=88.8% | precision=0.872 | recall=0.962 | F1=0.915

AUC=0.943 | PR-AUC=0.953

Matriz de confusión:

```
 [[179  55]
 [ 15 375]]
```

Reporte por clase:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Sano(0) | 0.92 | 0.76 | 0.84 | 234 |
| Neumonía(1) | 0.87 | 0.96 | 0.91 | 390 |
| accuracy | | | 0.89 | 624 |
| macro avg | 0.90 | 0.86 | 0.88 | 624 |
| weighted avg | 0.89 | 0.89 | 0.89 | 624 |

## Conclusiones:

Los seis modelos mantuvieron un **buen rendimiento global (Acc > 86 %, F1 > 0.89, AUC > 0.91)**, confirmando que la red quanvolucional logra generalizar en el conjunto de test.
Sin embargo, los comportamientos difieren por configuración:

- **Modelos 3 (A3 y B3)** alcanzaron las métricas más altas (F1≈0.92, AUC≈0.94) pero con **caída notoria entre validación y test**, señal de **sobreentrenamiento leve**.
  Las curvas de pérdida y la variabilidad del umbral *thr*∗ respaldan esta conclusión. Son modelos "ambiciosos", i. e. aprenden rápido, pero sacrifican estabilidad.

- **Modelos 2 (A2 y B2)** muestran **el aprendizaje más estable y conservador**, con diferencias mínimas entre validación y test (≈1–2 puntos).
  No presentan sobreajuste y podrían **seguir mejorando con más épocas**.
  Son los más confiables clínicamente y los que **mejor generalizan**.

- **Modelos 1 (A1 y B1)** logran un **equilibrio entre ambos extremos**: métricas altas, pérdida controlada y bajo riesgo de sobre entrenamiento gracias a su mayor regularización (WD=1e−3).

Considero que la reducción beneficia bastante en estos casos de CPU, pues si bien los modelos B tuvieron casi el doble más de épocas, tardaron 2 veces menos y con resultados muy cercanos.

## Anexo

**Tabla 18. Flujo de la red completa.**

| Paso | Versión A – sin reducción (28×28) | Versión B – con reducción (AvgPool 28→14) |
|---|---|---|
| **1. Entrada** | x ∈ ℝ^{B,1,28,28} | |
| **2. Preprocesamiento clásico** | Sin pooling; se trabaja a 28×28 | AvgPool2d(2,2) → x_red ∈ ℝ^{B,1,14,14} |
| **3. División en parches 2×2** | Opción usada: stride=2 → grid 14×14 ⇒ 196 parches | Sin traslape en 14×14 ⇒ grid 7×7 ⇒ 49 parches |
| **4. Representación del parche** | Cada parche p ∈ ℝ^{2×2} → x_p ∈ ℝ^4 (aplanado) | Igual: x_p ∈ ℝ^4 |
| **5. Codificación a ángulos** | x_p[w] ∈ [0,1] → φ_w = π·x_p[w] | |
| **6. Quanvolución (QV-1 / QV-k)** | QNode por parche (circuito HEA/ansatz breve) → C_q expectativas por parche. QV-1: C_q=1. QV-k: C_q=k. Salida por imagen: mapa de tamaño (H_p, W_p, C_q) con H_p=W_p=14. | Igual mecanismo; salida (7, 7, C_q) |
| **7. Ensamble tensorial** | Reordenado a ℝ^{B, C_q, H_p, W_p} | Reordenado a ℝ^{B, C_q, 7, 7} |
| **8. Flatten** | D = H_p·W_p·C_q → A: D = 196·C_q | B: D = 49·C_q |
| **9. MLP-A (compacta)** | D → h → 1 (ReLU/SiLU; dropout opcional) | |
| **10. MLP-B (más capaz)** | D → h1 → h2 → 1 | |
| **11. Logits → probas** | ŷ = sigmoid(logits) ∈ ℝ^{B,1} | |
| **12. Pérdida (train/val)** | BCEWithLogitsLoss (+ pos_weight por desbalance). Se registra val_loss, AUC, F1. | |
| **13. Umbralización** | Dos vistas: thr=0.5 y thr* (barrido en val para F1/Youden). | |
| **14. Backprop** | Autograd en MLP/Flatten; adjoint en QNode para ∂Loss/∂θ_q (sin shots). | |
| **15. Test (final)** | Cargar best_state (de val) y evaluar en 0.5 y thr*. | |