Quanvolution 2×2 en Digits data set

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.
En el presente documento encontramos la explicación a detalle del pipeline híbrido llamado "**Quanvolution2x2_Digits8x8_PyTorch_PennyLane_binario**", donde usamos **PyTorch** y **PennyLane**.
aplicamos una **Quanvolution 2×2** en el conjunto **Digits** para un problema de 2 clases. Esta **Quanvolution** actúa como kernel compartido utilizando únicamente **4 qubits** con un **ansatz** (RY/RZ + CNOT-ring). Con Flatten + MLP (64→32→1) y BCEWithLogits + Adam, el modelo alcanzó **95.83% de accuracy en test y F1≈0.959 por clase en 5 épocas.** El enfoque conserva localidad y compartición de pesos con solo **24 parámetros cuánticos**, favoreciendo eficiencia y generalización.

Palabras clave: Variational Quantum Algorithm, Digits, Quanvolution, PyTorch, PennyLane.

1) Introducción.

1.1. ¿Qué buscamos hacer?

Se busca entrenar un clasificador binario para imágenes obtenidas del **Digits data set (8x8)**, lo cual combine 2 etapas (Parte clásica y parte cuántica).

Estos sistemas son llamados híbridos (clásico/cuántico), una parte del pipeline corre con tensores y capas clásicas (**PyTorch**) y la otra parte es un circuito cuántico variacional (**PennyLane**). Tanto los parámetros cuánticos θ y clásicos ψ, se actulizan juntos con el mismo optimizador (**Adam**).

En este documento se presenta el desarrollo matemático de lo que realiza el código, más que nada para comprender como se van transformando lo datos y ver los cambios dimensionales.

1.2. Quanvolution 2×2.

Es una capa "tipo conv" en la que el kernel es un **circuito cuántico variacional** (PQC) que se reutiliza (comparte parámetros) sobre cada parche local de la imagen. Su funcionamiento consiste en las siguientes etapas:

a) **Codificación:** Mapeamos intensidades de píxeles 2×2 a ángulos de rotación para 4 qubits, transformando datos clásicos a estados cuánticos.
b) **Ansatz:** Aplicamos rotaciones entrenables y entrelazamiento (CNOTs) para crear correlaciones cuánticas entre píxeles mediante lo que se le conoce como Ansatz anillo.
c) **Rotaciones entrenables:** Parámetros del circuito que el modelo optimiza durante el entrenamiento.
d) **Entrelazamiento:** Se Establecen las correlaciones cuánticas que capturan relaciones no lineales entre píxeles.
e) **Compartición de pesos:** El mismo circuito cuántico se aplica a todos los parches de la imagen.
f) **Medición:** Obtenemos 4 valores reales [-1,1] midiendo observables Z en cada qubit, formando los canales de salida.
g) **Rearmado espacial:** Recolocamos las salidas en sus posiciones originales para construir el mapa de características 4×4×4.
h) **Cabeza clásica:** Red neuronal clásica que procesa los mapas cuánticos para producir la predicción final.

Podemos notar que como estamos usando parches 2×2 para asignar un qubit por pixel, si en algún proyecto futuro queremos usar las misma idea con parches de 4×4 haremos uso de 16 qubits. La ventaja en esto es que podemos reutilizar el parche como si se tratase de un kernel, lo cual lo hace escalable. En otras palabras, el coste cuántico depende del tamaño del parche, no del tamaño total de la imagen.

En un trabajo realizado anteriormente trabaje con 16 qubits tras una reducción de $8 \times 8 \rightarrow 4 \times 4$ del mismo data set, con un Ansatz HEA 2D brickwork. Se espera que usando estos sistemas pueda ser escalable el proyecto a imágenes más grandes.

1.3. Entrenamiento: pérdida, logits y backprop.

La cabeza clásica producirá un logit ℓ. Haciendo uso de una sigmoide calcularemos la probabilidad de clase. Por otro lado, tendremos a BCEWithLogitsLoss el cual combina sigmoide + BCE de modo numéricamente estable. En el backprop, PyTorch deriva la parte clásica; PennyLane aporta $\frac{\partial \langle Z \rangle}{\partial \theta}$ del QNode; el optimizador Adam actualiza θ y ψ.

2) Configuración experimento.

2.1. Selección de clases. (tarea binaria)

El código permite seleccionar los dos dígitos que vamos a distinguir, alguno es asignado como positivo (1) y el otro como negativo (0). La idea de realizarlo de esta forma es para ir entendiendo los demás conceptos, desarrollamos el problema lo

más simple que podamos para tener más claro la matemática del programa, lo cual es lo que nos interesa. En futuros proyectos se espera poder mejorarlo para poder comparar mayor número de clases, por el momento se plantea un problema binario.

2.2. Hiperparámetros de entrenamiento.

Utilizamos **Batches** chicos más que nada por el tamaño del data set. Estos lotes recordemos que son las imágenes que serán procesadas por cada actualización.

Las **épocas** son las pasadas completas por el conjunto de entrenamiento.

Decidí un **learning rate** de 3e-3, este nos dice qué tan grande es cada actualización. Usando este learning rate con Adam suele despegar rápido.

Para la **regularización** he decidido usar $L2$, así regularizando los pesos clásicos para evitar lo máximo posible un sobre ajuste. Este parámetro sería mucho a tomar en cuenta para futuros proyectos más grandes, donde habría que estar más atento al sobre ajuste.

2.3. Parámetros de la capa cuántica. (Quanvolution)

Definimos el tamaño del parche local que alimenta al kernel cuántico. El número de qubits depende directamente del parche, en este caso usamos un qubit por píxel del parche. Además, también se agrega un parámetro para la profundidad del ansatz, donde entre más capas se tenga habrá mejores correlaciones, sin embargo, implica el riesgo de entrenamientos lentos o gradientes pequeños.

2.4. Simulador.

Utilizo DIFF_METHOD para poder asignar el método de diferenciación, sugiero utilizar las siguientes 2 opciones:

a) "**adjoint**": método eficiente para simulador (statevector). Rápido y estable.
b) "**parameter-shift**": regla agnóstica a hardware, compatible con shots. Es más lento, pero sería lo mejor para aproximar el comportamiento en un dispositivo real.

Por otro lado, para los SHOTS se sugieren igual 2 opciones:

a) **None**: sin muestreo, la medición es determinista (más rápido).
b) **Entero (p.ej. 200, 1000)**: se aproxima la expectativa por repetición de mediciones; introduce ruido estocástico (más realista y a veces regulariza).

Cambiando estos parámetros podemos estudiar cómo responde nuestro modelo a esas diferentes circunstancias y usar lo aprendido para aplicarlo a futuros códigos.

Se presentan otras 2 opciones:

a) **Usar GAP**: Resume cada canal a un número (promedio espacial). Es compacto y robusto, pero puede perder detalle lo que lleva que a veces el modelo se estanca.
b) **No usar GAP**: conserva los 64 valores (4 canales × 4×4), esto da más capacidad a la cabeza clásica y a despegar el aprendizaje.

Finalmente usamos una salida 1 logit para binario, siendo más estable con BCEWithLogitsLoss.

3) Carga de datos.

3.1. Digits. (Dataset, Máscaras y Normalización)

El dataset es una colección de imágenes $\{X_i\}_{i=1}^{N}$, con $X_i \in \mathbb{R}^{8\times 8}$ y etiquetas $\{y_i\}_{i=1}^{N}$, con $y_i \in \{0, 1, \dots, 9\}$. Por lo tanto, trabajamos con las dimensiones:

$$X \in \mathbb{R}^{N\times 8\times 8} \ \wedge \ y \in \{0, 1, \dots, 9\}^{N}$$

Con codificación numérica: float32 para entradas e int64 para etiquetas.

Vamos a solucionar un problema binario sobre dos clases elegidas: c_+ **(positiva)** y c_- **(negativa)**. La máscara:

$$m_i = 1\{y_i = c_+ \text{ o } y_i = c_-\} \in \{0,1\}$$

Al **filtrar**, nuestro espacio se reduce a:

$$\tilde{X} = \{X_i\}_{i\in\mathcal{I}} \ \wedge \ \tilde{y} = \{y_i\}_{i\in\mathcal{I}}, \qquad \mathcal{I} = \{i : m_i = 1\}$$

De tamaño $\tilde{N} = |\mathcal{I}|$.

Definiendo nuestro **Re-etiquetado** a $\{0, 1\}$ de la siguiente forma:

$$\hat{y}_i = \begin{cases} 1, & y_i = c_+ \\ 0, & y_i = c_- \end{cases} \Rightarrow \hat{y} \in \{0,1\}^{\tilde{N}}$$

En Digits, cada pixel p satisface $p \in \{0,1, \dots ,16\}$. **Normalizamos**:

$$\bar{X}_i = \frac{X_i}{16} \in [0,1]^{8\times 8}$$

Esta normalización es clave porque el **feature map cuántico** usa $\phi = \pi x$; al tener $x \in [0,1]$, obtenemos $\phi \in [0, \pi]$de forma acotada y estable.

3.2. PyTorch. (Tensores y canales)

Procedemos a convertir nuestros datos a tensores con **PyTorch**:

$$X_t \in \mathbb{R}^{\tilde{N}\times 1\times 8\times 8}, \qquad y_t \in \{0,1\}^{\tilde{N}}$$

Usamos **unsqueeze (1)**, para agregar la dimensión de canal $C = 1$(formato NCHW) que usan las capas tipo conv/unfold.

$$\text{shape}(X_t) = (\tilde{N}, \underset{\breve{C}}{1}, 8, 8).$$

3.3. Partición estratificada. (Train, Val y Test)

Definimos tres subconjuntos disjuntos:

- **Test**: fracción 0.2 de los datos (≈ 20%).
- Del restante 80%, se separa **Validación**: 0.2 de ese bloque (≈ 16% del total), quedando **Train** ≈ 64%.

Utilizamos estratificación preservar la proporción de clases en cada subconjunto, para evitar sesgos.

3.4. DataLoaders y mini-batching.

Definimos las **medidas empíricas** sobre cada **split**:

$$\hat{P}_{\text{train}} = \frac{1}{N_{\text{tr}}} \sum_{i \in \text{train}} \delta_{(X_i, y_i)}$$

Análogo para $\hat{P}_{\text{val}}$ y $\hat{P}_{\text{test}}$.

Tenemos **mini-batching** de tamaño B, los cuales son subconjuntos $\mathcal{B} \subset$ train y nos sirve para optimizar por Descenso Estocástico (SGD/Adam) sobre el riesgo empírico por batch:

$$\mathcal{L}_{\mathcal{B}}(\Theta) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \ell(f_\Theta(X_i), y_i)$$

Finalmente se hace uso de **shuffle=True** (solo en train) para aleatorizar el orden de $\mathcal{B}$ por época, lo que mejora la mezcla y el comportamiento estocástico del optimizador.

4) Dispositivo cuántico, QNode (Quanv) y capa Quanv2x2

4.1 Dispositivo cuántico. (simulador)

Utilizamos PennyLane para crear nuestro dispositivo, el cual contara con $Q = 4$ wires (Qubits en PennyLane), pues tenemos parches de 2×2, i. e. un qubit por pixel. Así que estaremos trabajando para el espacio de Hilbert $\mathcal{H} = (\mathbb{C}^2)^{\otimes Q}$de dimensión 2^Q y tenemos nuestro estado inicial es $|\psi_0\rangle = |0\rangle^{\otimes Q}$, pues tenemos nuestros qubits inicializados en $|0\rangle$.

No hay necesidad usar lightning.qubit, pues estamos trabajando con poco qubits por lo que default.qubit da excelentes resultados. Puede que en futuros trabajos deba realizarlos con lightning.qubit, para trabajar con imágenes más grandes y crear parches aún mayores.

Por otro lado, podemos cambiar también el parámetro de shots, ya sea si se deciden expectativas exactas (como fue en este caso para los resultados mostrados) o bien modificar la expectativa $\langle O \rangle$ por media muestral $\hat{z} = \frac{1}{m} \sum_{k=1}^{m} z^{(k)}$ con varianza $\leq 1/m$ para observables acotados en $[-1,1]$.

4.2 QNode: Q_θ: $\mathbb{R}^4 \to [-1, 1]^4$.

El QNode implementa un mapeo **diferenciable** por parche:

$$\phi \in \mathbb{R}^4 \quad \overset{Q_\theta}{\mapsto} \quad \text{z} = (z_0, z_1, z_2, z_3) \in [-1,1]^4.$$

Consta de tres etapas: **embedding**, **ansatz** y **medición**.

- **dev**: Es el dispositivo cuántico donde corre el circuito. (Creado arriba)
- **interface="torch"**: Le dice al QNode que su entrada y salida sean tensores PyTorch y que el gradiente fluya a través del autograd de PyTorch.
- **diff_method=DIFF_METHOD**: Define como calcular los gradientes del circuito cuántico:
    - "adjoint": método adjunto (rápido en simulador de estado). No usa parameter-shift; propaga el gradiente "de una" hacia atrás. Ideal para prototipado sin shots.
    - "parameter-shift": regla de desplazamiento de parámetros (válida en hardware y con shots). Calcula derivadas evaluando el circuito dos veces por parámetro ($\pm\pi/2$). Más lento, pero agnóstico a hardware.
    - (Otros posibles: "backprop" en ciertos dispositivos, "finite-diff" para diferencias finitas, etc., pero los comunes aquí son los dos de arriba).

En resumen, lo que el decorador le hace al conjunto es que envuelve el circuito cuántico para ejecutarlo en el dispositivo especificado y obtener expectativas como tensores PyTorch, mientras registra el grafo computacional para permitir la diferenciación automática y la retropropagación de gradientes.

A. Embedding. (feature map)

Preparamos el estado inicial $|0\rangle^{\otimes 4}$ y aplicamos codificación angular:

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

Con $\phi_w = \pi x_w,\ x_w \in [0,1]$ como lo comentamos en la sección 3.1. Pues se garantiza que el dominio está acotado en $[0,1]$. Recordemos que es lo que sucede al aplicar la matriz R_Y:

$$|\psi_{\text{emb}}\rangle = \left(\bigotimes_{w=0}^{3} R_Y(\phi_w) \right) |0\rangle^{\otimes 4} = \bigotimes_{w=0}^{3} \left(\cos\frac{\phi_w}{2} |0\rangle_w + \sin\frac{\phi_w}{2} |1\rangle_w \right)$$

Lo cual es un producto de qubits "inclinados" en el plano Y sí lo viéramos en la representación de Bloch; aún **sin entrelazamiento,** pero ya están en superposición.

B. Ansatz variacional

Este **Ansatz** lo podemos dividir en 2 secciones; una primer parte de rotaciones $R_Y(\theta)$ y $R_Z(\theta)$, y, una segunda parte de entrelazamiento de qubits.

Nos encontramos con el parámetro **weights**, el cual tiene la forma $(L, Q, 2)$: por capa $\ell \in \{1, \dots, L\}$ del ansatz (los layers) y qubit w, así que **hay dos ángulos** $\theta_{\ell,w}^{(y)}$y $\theta_{\ell,w}^{(z)}$. A estos últimos 2 son a los que les asignamos las rotaciones. Nuestro entrelazador es uno tipo anillo:

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

Así que el Ansatz para **una capa ℓ** se lee como:

I. Para cada *wire* w: RY(weights[l,w,0]) luego RZ(weights[l,w,1]).
II. Sespués aplicamos el anillo de CNOTs.

$$U_\ell(\theta_\ell) = E \left[\bigotimes_{w=0}^{3} R_Z\left(\theta_{\ell,w}^{(z)}\right) R_Y\left(\theta_{\ell,w}^{(y)}\right) \right]$$

El **estado de salida** para un parche queda entonces:

$$|\psi_{\text{out}}\rangle = \left(\prod_{\ell=1}^{L} U_\ell(\theta_\ell) \right) U_{\text{emb}}(\phi) |0\rangle^{\otimes 4}, \quad U_{\text{emb}}(\phi) = \bigotimes_{w} R_Y(\phi_w)$$

Por ejemplo.
Para $L = 3$ capas y trabajando con $w = 0,1,2,3,4$:

$$\boxed{|\psi_{\text{out}}\rangle = U_3(\theta_3)U_2(\theta_2)U_1(\theta_1)U_{\text{emb}}(\phi)|0\rangle^{\otimes 4}}$$

Embedding (datos):

$$U_{\text{emb}}(\phi) = R_Y^{(0)}(\phi_0) \otimes R_Y^{(1)}(\phi_1) \otimes R_Y^{(2)}(\phi_2) \otimes R_Y^{(3)}(\phi_3)$$

Capa $\ell = 1$: $U_1(\theta_1)$

$$U_1(\theta_1) = E\left(R_Z^{(0)}(\theta_{1,0}^{(z)})\, R_Y^{(0)}(\theta_{1,0}^{(y)}) \otimes R_Z^{(1)}(\theta_{1,1}^{(z)})\, R_Y^{(1)}(\theta_{1,1}^{(y)}) \otimes R_Z^{(2)}(\theta_{1,2}^{(z)})\, R_Y^{(2)}(\theta_{1,2}^{(y)}) \otimes R_Z^{(3)}(\theta_{1,3}^{(z)})\, R_Y^{(3)}(\theta_{1,3}^{(y)})\right)$$

Capa $\ell = 2$: $U_2(\theta_2)$

$$U_2(\theta_2) = E\left(R_Z^{(0)}(\theta_{2,0}^{(z)})\, R_Y^{(0)}(\theta_{2,0}^{(y)}) \otimes R_Z^{(1)}(\theta_{2,1}^{(z)})\, R_Y^{(1)}(\theta_{2,1}^{(y)}) \otimes R_Z^{(2)}(\theta_{2,2}^{(z)})\, R_Y^{(2)}(\theta_{2,2}^{(y)}) \otimes R_Z^{(3)}(\theta_{2,3}^{(z)})\, R_Y^{(3)}(\theta_{2,3}^{(y)})\right)$$

Capa $\ell = 3$: $U_3(\theta_3)$

$$U_2(\theta_2) = E\left(R_Z^{(0)}(\theta_{3,0}^{(z)})\, R_Y^{(0)}(\theta_{3,0}^{(y)}) \otimes R_Z^{(1)}(\theta_{3,1}^{(z)})\, R_Y^{(1)}(\theta_{3,1}^{(y)}) \otimes R_Z^{(2)}(\theta_{3,2}^{(z)})\, R_Y^{(2)}(\theta_{3,2}^{(y)}) \otimes R_Z^{(3)}(\theta_{3,3}^{(z)})\, R_Y^{(3)}(\theta_{3,3}^{(y)})\right)$$

Podemos apreciar claramente los 4 qubits por capa.

Parámetros entrenables.

$$\Theta = \left\{\theta_{\ell,w}^{(y)}, \theta_{\ell,w}^{(z)} \mid \ell \in \{1,2,3\} \ \wedge \ w \in \{0,1,2,3\}\right\}$$

Es decir, 24 parámetros cuánticos en total (3capas × 4qubits × 2ángulos por qubit y capa).

Ángulos de embedding (provenientes del parche 2×2 normalizado)

$$\phi = (\phi_0, \phi_1, \phi_2, \phi_3), \qquad \phi_w = \pi\, x_w$$

Donde $x_w \in [0,1]$ son las intensidades de los pixeles.

C. Expectativas.

Estado final del circuito para un parche:

$$|\psi_{\text{out}}\rangle = U(\theta)\, U_{\text{emb}}(\phi)|0\rangle^{\otimes 4}$$

Aplicamos una **lectura** de 4 canales (uno por qubit):

$$z_w = \langle \psi_{\text{out}} \mid Z_w \mid \psi_{\text{out}} \rangle = p_w(0) - p_w(1) \in [-1,1], \ \ w = 0,1,2,3$$

Al medir un qubit en la base Z se modela como una **medición proyectiva**: aplicanado los proyectores $P_0 = |0\rangle\langle 0|$ o $P_1 = |1\rangle\langle 1|$ con probabilidades dadas por la regla de Born, y el estado **colapsa** al subespacio correspondiente.

Cuando pedimos expval(PauliZ), lo que obtenemos es **la esperanza** $\langle Z\rangle \in [-1,1]$: un promedio ideal de muchos resultados $(+1, -1)$.

En simulación con shots=None **no se simulan colapsos**; el valor se calcula directamente del statevector. En cambio, con shots>0 (o en hardware real) sí hay **muchas mediciones proyectivas** independientes: cada shot colapsa el estado de ese experimento, se **reprepara** el circuito desde $|0\rangle^{\otimes Q}$, y la media de todos los ± 1 aproxima $\langle Z\rangle$.

Así, el "kernel cuántico compartido" implementa:

$$Q_\theta(\phi) = (z_0, z_1, z_2, z_3) \in [-1,1]^4.$$

4.3. Capa Quanv2x2.

La idea general de la clase es **aplicar un único kernel cuántico Q_θ** (con parámetros compartidos θ) a cada parche 2×2 de cada imagen del batch y re-armar un mapa de características de tamaño $4 \times 4 \times 4$ (4 canales × 4 filas × 4 columnas). Por "**parámetros compartidos**" entendemos que el mismo conjunto θ se reutiliza en todas las posiciones y todas las imágenes, exactamente como un kernel convolucional clásico.

Extracción de parches como operador lineal (im2col)

Sea una imagen $X \in \mathbb{R}^{1\times 8\times 8}$. Definimos el operador lineal

$$\mathcal{U}_{\text{unfold}}: \mathbb{R}^{1\times 8\times 8} \longrightarrow \mathbb{R}^{4\times 16}$$

Vemos que devuelve las **16 ventana**s (parches) no solapadas (ordenadas) de **tamaño 2×2**, con cada una aplanada a 4 componentes.

En notación matricial, si vectorizamos X a $\text{vec}(X) \in \mathbb{R}^{64}$, existe una matriz binaria $M \in \{0,1\}^{(4\cdot 16)\times 64}$ tal que:

$$\text{vec}(P) = M\,\text{vec}(X), \qquad P \in \mathbb{R}^{4\times 16}.$$

F.unfold implementa esta transformación (para todo el batch a la vez).

Feature map clásico → ángulos del circuito

Definimos $\phi = \pi\, \text{v}$, para normaliza cada pixel que para cada parche $\text{v} \in [0,1]^4$. Esto acota $\phi_w \in [0,\pi]$, manteniendo al embedding dentro de una región estable del Bloch. Así, estos valores podrán entrar al Feature map cuántico, como lo vimos en una sección pasada.

$$(v_0, v_1, v_2, v_3) \longrightarrow (\phi_0, \phi_1, \phi_2, \phi_3)$$

Aplicación del kernel cuántico compartido a cada parche

ara cada fila $\phi_{b,\ell}$(parche ℓde la imagen b), evaluamos

$$\text{z}_{b,\ell} = \ Q_\theta(\phi_{b,\ell}) \in [-1,1]^4.$$

Aquí, θ es el mismo para todos los parches y todas las imágenes (parámetros compartidos), análogo a un filtro de una CNN. Durante el entrenamiento, los gradientes contribuyen aditivamente desde todas las posiciones:

$$\frac{\partial \mathcal{L}}{\partial \theta} \ = \ \sum_b \sum_{\ell=0}^{15} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_{b,\ell}} \ \frac{\partial \mathbf{z}_{b,\ell}}{\partial \theta}$$

Rearmado espacial (*col2im*) al mapa 4×4×4

Denotemos por $Z_b \in \mathbb{R}^{4\times 16}$ la matriz de salidas por parche de la imagen b (4 canales × 16 posiciones). Debemos **reubicar** cada columna (posición de parche) en su **coordenada** (i,j) de la rejilla 4×4. Si numeramos las posiciones por

$$\ell \ = \ 4\,i + j, \qquad i,j \in \{0,1,2,3\},$$

entonces el **mapa cuántico** $\mathcal{Z}_b \in \mathbb{R}^{4\times 4\times 4}$queda definido por

$$\mathcal{Z}_b[c,i,j] = Z_b[c, \ell = 4i + j], \qquad c \in \{0,1,2,3\}.$$

Formalmente, hacemos el reordenamiento lineal $\mathcal{R}$ (permuta/reshape) tal que $\mathcal{Z}_b = \ \mathcal{R}(Z_b) \in \mathbb{R}^{4\times 4\times 4}$.

Tipo de dato y consistencia (para PyTorch)

Finalmente, no aseguramos de que $\mathcal{Z}$ tenga el mismo **dtype** que el *input* (evitar conflictos con capas densas en PyTorch: float32 vs float64).

5) Cabeza clásica y ensamblaje del modelo

5.1. Clase HeadSmall: del mapa cuántico al logit

Tomamos el **mapa cuántico** que sale de la Quanv2×2

$$\mathcal{Z} \in \mathbb{R}^{B\times 4\times 4\times 4}$$

(4 canales, rejilla 4×4 por imagen) y lo transformamos en un **logit** $\ell \in \mathbb{R}^{B\times 1}$. Ese logit se usa con **BCEWithLogitsLoss** durante el entrenamiento (la sigmoide se aplica implícitamente dentro de la pérdida). Para ello tenemos 2 caminos:

A) Global Average Pooling (Con GAP)

Promediamos cada canal sobre la rejilla:

$$g_b[c] \;=\; \frac{1}{4\cdot 4}\sum_{i=0}^{3}\sum_{j=0}^{3} \mathcal{Z}_b[c,i,j] \Rightarrow g_b \in \mathbb{R}^4,$$

y luego una capa lineal $4 \to 1$ produce ℓ_b. Proyección lineal a 1 logit: $\ell_b \;=\; w^{\top} g_b + b, \ell_b \in \mathbb{R}$.

B) Flatten (con pequeña MLP – Sin GAP)

Aplanamos todo el mapa: $f_b = \mathrm{vec}(\mathcal{Z}_b) \in \mathbb{R}^{64}$, pasamos por una capa oculta con no linealidad. Capa oculta + ReLU:

$$h_b = \sigma(W_1 f_b + b_1) \in \mathbb{R}^{32}, \qquad \sigma = \mathrm{ReLU}.$$

Aplicamos Dropout (regularización) en entrenamiento, lo que anula aleatoriamente fracciones de h_b. Así nuestro logit final para este caso:

$$\ell_b = W_2^{\top} h_b + b_2 \;\in\; \mathbb{R}.$$

Importancia:

Por qué:

- **Sin GAP:** mantiene la estructura espacial del mapa (4×4×4) y da capacidad para combinar patrones locales, lo que hace que suela aprender más rápido y llegar más alto. Parámetros extra ≈ **2,113**, manejables con Dropout y weight_decay. Podríamos usarlo en clases difíciles, simulación determinista (shots=None), o con la necesaidad subir accuracy.
- **Con GAP:** comprime mucho (y se pierde dónde ocurrió el patrón). Si bien es robusto al ruido, puede estancarse en tareas donde la posición importa. Convendría activarlo con shots o si buscamos un modelo mínimo y muy estable/regularizado.

En ambos modos, la **salida** de HeadSmall es un **logit** $\ell \in \mathbb{R}^{B\times 1}$.

5.2. Clase QuanvModel: ensamblaje Quanv + cabeza

QuanvModel conecta la Quanvolution 2×2 (que produce un mapa cuántico) con la cabeza clásica (que convierte ese mapa en un logit binario). Vamos a entender el flujo repasando lo que hemos visto en secciones anteriores.

La entrada es un batch de imágenes normalizadas $x \in \mathbb{R}^{B\times 1\times 8\times 8}$. Primero, Quanv2x2 divide cada imagen en **16 parches 2×2**, aplica a **cada parche** el **mismo circuito cuántico variacional** $\mathcal{Q}_\theta$ (parámetros **compartidos** θ), y **mide** 4 expectativas $\langle Z\rangle \in [-1,1]^4$ (una por qubit). Esas 16 salidas por canal se **rearman** en su posición para obtener el **mapa cuántico** $\mathcal{Z} \in \mathbb{R}^{B\times 4\times 4\times 4}$(4 canales × rejilla 4×4). Después, HeadSmall transforma $\mathcal{Z}$ en un **logit** $\ell \in \mathbb{R}^{B\times 1}$ mediante uno de dos caminos: **(i)** GAP o **(ii)** Flatten + MLP. El logit ℓ se usa con BCEWithLogitsLoss.

6) Entrenamiento y validación.

6.1. Criterio de pérdida y configuración del optimizador

Como se trata de clasificación binaria utilizamos **BCEWithLogits** por su estabilidad numérica. Está opera directamente sobre el **logit** ℓ y aplica la sigmoide internamente. Para una muestra b con etiqueta $y_b \in \{0,1\}$, la pérdida es

$$\mathcal{L}_{\text{BCE-Logits}}(\ell_b, y_b) = -\big[y_b \log \sigma(\ell_b) + (1-y_b)\log\big(1-\sigma(\ell_b)\big)\big],$$
$$\sigma(\ell) = \frac{1}{1+e^{-\ell}}.$$

En un mini-batch de tamaño B, se promedia sobre $b = 1, \dots, B$. Para la optimización empleamos **Adam**, que combina momento y escalado adaptativo por parámetro, con **L2 (weight decay)** sobre los pesos clásicos para mitigar sobreajuste. Un **scheduler** del tipo ReduceLROnPlateau reduce la tasa de aprendizaje cuando la pérdida de validación deja de mejorar.

6.2 Métrica de desempeño: exactitud (accuracy)

Durante el entrenamiento y la validación se va a ir reportando el **accuracy** por mini-batch. Dado el logit ℓ_b, la predicción binaria se obtiene con umbral en **0** (equivalente a umbral 0.5 sobre $\sigma(\ell_b)$):

$$\hat{y}_b = \mathbf{1}\{\ell_b \geq 0\}.$$

La exactitud del batch es $\frac{1}{B}\sum_b \mathbf{1}\{\hat{y}_b = y_b\}$.

6.3. Lazo de entrenamiento y validación por época

En cada época, para cada mini-batch hacemos un **forward** que lleva $x \in \mathbb{R}^{B\times 1\times 8\times 8}$ a mapa cuántico $\mathcal{Z} \in \mathbb{R}^{B\times 4\times 4\times 4}$(Quanv 2×2) a logit $\ell \in \mathbb{R}^{B\times 1}$ (cabeza clásica).

Con la pérdida $\mathcal{L}_{\text{BCE-Logits}}(\ell, y)$ calculamos gradientes y actualizamos los dos grupos de parámetros:

- **Clásicos** ψ de la cabeza:
  - **GAP**: son el peso y sesgo de Linear(4→1).
  - **Flatten**, son $W_1 \in \mathbb{R}^{32\times 64}, b_1 \in \mathbb{R}^{32}, W_2 \in \mathbb{R}^{1\times 32}, b_2 \in \mathbb{R}$.
- **Cuánticos** θ del bloque Quanv: Estos están almacenados en quanv.weights con forma (LAYERS,QUBITS, 2) (en nuestro caso $3\times 4\times 2 = 24$): por capa y por *wire* hay dos ángulos entrenables para *RY* y *RZ*.

Con PyTorch usamos *autograd* sobre ψ, y PennyLane provee $\partial\mathcal{Z}/\partial\theta$ para actualizar θ usando el método elegido del QNode (adjoint en simulación o parameter-shift si quieres compatibilidad

con *shots*/hardware). Con Adam, ambos conjuntos (ψ, θ) se optimizan conjuntamente.

NOTA: Si deseas conocer más sobre como se llevan estas actualizaciones te invito a leer mi anterior documento "Ansatz HEA 2D brickwork para VQA", donde explico a más detalle.

Gradiente.
Sea un mini-batch de tamaño B. Para cada imagen b, su logit ℓ_by etiqueta $y_b \in \{0,1\}$. Definimos el "error" estándar de BCE con logits:

$$\delta_b = \frac{\partial \mathcal{L}}{\partial \ell_b} = \sigma(\ell_b) - y_b, \qquad \sigma(\ell) = \frac{1}{1+e^{-\ell}}.$$

Cada imagen produce un **mapa cuántico** $Z_b \in \mathbb{R}^{4\times4\times4}$. Indexamos canales $c \in \{0,1,2,3\}$ y posiciones $(i,j) \in \{0,1,2,3\}^2$ con el índice de parche $\ell = 4i + j$. Denotamos

$$z_{b,c\ell} \equiv Z_b[c,i,j] = \langle Z_c \rangle.$$
(salida del QNode para el parche ℓ, canal c)

Sea $\Theta = (\theta, \psi)$el conjunto **total** de parámetros (cuánticos θ y clásicos ψ). Entonces, el **gradiente total** de la pérdida es:

$$\frac{\partial \mathcal{L}}{\partial \Theta} = \sum_{b=1}^{B} \left[\underbrace{\sum_{\ell=0}^{15} \sum_{c=0}^{3} \delta_b \frac{\partial \ell_b}{\partial z_{b,c\ell}} \frac{\partial z_{b,c\ell}}{\partial \theta}}_{\frac{\partial \mathcal{L}}{\partial \theta}} \oplus \underbrace{\frac{\partial \ell_b}{\partial \psi} \delta_b}_{\frac{\partial \mathcal{L}}{\partial \psi}} \right]$$

Vemos que el factor $\frac{\partial \ell_b}{\partial z_{b,c\ell}}$ depende solo de la cabeza (GAP o Flatten+MLP). Por otro lado, el factor $\frac{\partial z_{b,c\ell}}{\partial \theta}$ es el gradiente cuántico del observable (se obtiene con parameter-shift o adjoint).
La suma en b, ℓ, crefleja que θ es **compartido** por **todos** los parches y **todas** las imágenes.

Parámetros cuánticos θ (bloque Quanv):

Parámetros entrenables del ansatz (en el código: quanv.weights):
$\theta = \left\{ \theta^{(y)}_{\ell',w}, \theta^{(z)}_{\ell',w} | \ell' = 1, \dots, L,\ w = 0, \dots, 3 \right\}$(con $L = 3 \Rightarrow 24$ params).
Gradiente total respecto a cualquier $\vartheta \in \theta$:

$$\frac{\partial \mathcal{L}}{\partial \vartheta} = \sum_{b=1}^{B} \sum_{\ell=0}^{15} \sum_{c=0}^{3} \delta_b \frac{\partial \ell_b}{\partial z_{b,c\ell}} \frac{\partial z_{b,c\ell}}{\partial \vartheta}$$

El término **cuántico** $\partial z / \partial \vartheta$ se obtiene de dos maneras equivalentes:

- **Parameter-shift** (exacto para R_Y, R_Z):
$$\frac{\partial z_{b,c\ell}}{\partial \vartheta} = \frac{1}{2}\left(z_{b,c\ell}\,|_{\vartheta+\frac{\pi}{2}} - z_{b,c\ell}\,|_{\vartheta-\frac{\pi}{2}}\right).$$
- **Adjoint** (reverse-mode en simulador):
$$\frac{\partial z_{b,c\ell}}{\partial \vartheta} = \mathrm{Im}\,\langle \lambda_{k+1} \mid G_k \mid \psi_{k+1} \rangle$$
para la puerta: $U_k(\vartheta) = e^{-i\frac{\vartheta}{2}G_k}$.

Parámetros clásicos ψ(cabeza) - Flatten + MLP
Parámetros entrenables:

- $W_1 \in \mathbb{R}^{32\times64}$, $b_1 \in \mathbb{R}^{32}$,
- $W_2 \in \mathbb{R}^{1\times32}$, $b_2 \in \mathbb{R}$.
(El Dropout **no** tiene parámetros entrenables.)

Definiciones (para cada imagen b):

$$\mathrm{f}_b = \mathrm{vec}(Z_b) \in \mathbb{R}^{64},$$
$$\mathrm{a}_b = W_1 \mathrm{f}_b + b_1,$$
$$\mathrm{h}_b = \mathrm{ReLU}(\mathrm{a}_b) \in \mathbb{R}^{32},$$
$$\ell_b = W_2\, \mathrm{h}_b + b_2.$$

Gradientes de la segunda capa:

$$\frac{\partial \mathcal{L}}{\partial W_2} = \sum_{b=1}^{B} \delta_b\, \mathrm{h}_b^{\top}, \quad \frac{\partial \mathcal{L}}{\partial b_2} = \sum_{b=1}^{B} \delta_b$$

Retropropagación a la primera capa (usando la máscara de ReLU $M_b = \mathrm{diag}\,(\mathrm{a}_b > 0)$): $\mathrm{r}_b = M_b\, W_2^{\top}\, \delta_b \in \mathbb{R}^{32}$, ("error" en la capa oculta)

$$\frac{\partial \mathcal{L}}{\partial W_1} = \sum_{b=1}^{B} \mathrm{r}_b\, \mathrm{f}_b^{\top}, \quad \frac{\partial \mathcal{L}}{\partial b_1} = \sum_{b=1}^{B} \mathrm{r}_b$$

Sensibilidad de la cabeza respecto a cada **entrada** $z_{b,c\ell}$:

$$\frac{\partial \ell_b}{\partial z_{b,c\ell}} = \frac{\partial \ell_b}{\partial f_{b,k}} = (W_1^{\top} M_b\, W_2^{\top})_k \equiv s_{b,k}$$

Entonces, para parámetros cuánticos:

$$\frac{\partial \mathcal{L}}{\partial \vartheta} = \sum_{b=1}^{B} \sum_{\ell=0}^{15} \sum_{c=0}^{3} \delta_b\, s_{b,k(c,\ell)}\, \frac{\partial z_{b,c\ell}}{\partial \vartheta}$$

Para cada $\vartheta \in \theta$.

6.4. Selección del mejor modelo por validación

Para evitar quedarnos con una época que sobre ajustó, se conserva el estado con menor pérdida de validación a lo largo del entrenamiento. Al finalizar, se restaura ese state_dict antes de evaluar en prueba. Este procedimiento estabiliza el desempeño final y reduce la varianza entre corridas.

Con el modelo fijado, se evalúa sobre el conjunto de prueba sin gradientes (modo eval).

7) Resultados.

Los siguientes resultados mostrados contienen los parámetros en la tabla a continuación:

Parámetro	Valor
CLASS_POS	3
CLASS_NEG	8
BATCH_SIZE	32
EPOCHS	5
LR	3e-3
WEIGHT_DECAY	1e-4
PATCH_SIZE	2
QUBITS	4
LAYERS	3
DIFF_METHOD	adjoint
SHOTS	None
USE_GAP	False
Optimizer	Adam
Scheduler	ReduceLROnPlateau
Criterion	BCEWithLogitsLoss

Entrenamos el conjunto de entrenamiento y de validación:

Quanv dtype in/out: torch.float32 → torch.float32
[debug] ||grad(weights_quanv)|| = 1.603e-02

Época 01 | Train: loss=0.6304, acc=68.4% | Val: loss=0.5457, acc=86.0% | lr=3.00e-03

Época 02 | Train: loss=0.4963, acc=87.7% | Val: loss=0.4318, acc=91.2% | lr=3.00e-03

Época 03 | Train: loss=0.3720, acc=95.6% | Val: loss=0.3270, acc=94.7% | lr=3.00e-03

Época 04 | Train: loss=0.2762, acc=95.2% | Val: loss=0.2476, acc=98.2% | lr=3.00e-03

Época 05 | Train: loss=0.1925, acc=97.4% | Val: loss=0.1900, acc=96.5% | lr=3.00e-03

Podemos observar que la hay consistencia en los datos, aunque cabe resaltar que esa sección se agregó por errores que llegue a

tener por problemas numéricos. Con eso nos aseguramos de la consistencia de datos.

Por otro lado, decidí agregar también la norma de gradiente del bloque cuántico para confirmar que si hubiera señal de aprendizaje.

En 5 épocas se observa una mejora monótona de la pérdida y la exactitud tanto en **train** (68.4%→97.4%) como en **val** (86.0%→96.5%, con pico 98.2% en la época 4), lo que sugiere buena capacidad de generalización y sin sobreajuste temprano (la brecha train-val se mantiene pequeña).

La **LR** se mantuvo en 3e-3, suficiente para progresar de forma estable; con estas tendencias, es razonable continuar unas épocas más y conservar por checkpointing la mejor época (aquí, val=98.2% en la 4).

Aplicando el modelo al conjunto **test**, logramos un **95.83% de accuracy** con desempeño muy equilibrado entre clases: **F1≈0.959** para ambas.

Reporte de clasificación (0 = NEG, 1 = POS):

	precision	recall	f1-score	support
0	0.9211	1.0000	0.9589	35
1	1.0000	0.9189	0.9577	37
accuracy			0.9583	72
macro avg	0.9605	0.9595	0.9583	72
weighted avg	0.9616	0.9583	0.9583	72

- **Clase 0 (NEG):** recall 1.00 (no se escapó ningún NEG: 35/35 correctos) a costa de una precision 0.921 → hubo 3 falsos positivos (tres POS predichos como NEG).
- **Clase 1 (POS):** precision 1.00 (ningún falso positivo) con recall 0.919 → 3 falsos negativos (tres POS reales predichos como NEG).

Estos resultados confirman que la Quanvolution 2×2 + MLP generaliza bien y como siguiente paso, se propone probar SHOTS>0/parameter-shift para un escenario más realista, en un conjunto de datos más grande.

Anexo.

Algunas preguntas que pueden causar confusión:

¿8×8 pasa a 16 imágenes 2×2?

No se crean 16 imágenes nuevas separadas, se dividie la imagen 8×8 en 16 parches 2×2 (ventanas no solapadas) mediante unfold. Como si una rejilla 4×4 de pequeños bloques 2×2. Son trozos de la misma imagen y guardamos su posición.

¿Cómo entra cada parche al circuito?

Cada parche 2×2 (4 píxeles en [0,1]) se convierte en 4 ángulos. Esos 4 ángulos se aplican a 4 qubits (1 por píxel). Luego el ansatz (rotaciones entrenables + CNOTs) opera y medimos 4 expectativas ⟨Z⟩ → un vector de 4 "canales" para ese parche.

¿El ansatz recorre todos los datos?

Sí, la idea es reuitilizar los mismos parámetros para cada parche y para todas las imágenes del batch. I. e. un único kernel cuántico compartido aplicado muchas veces: 16 veces por imagen (una por parche). Los parámetros son los mismos en todas las posiciones; lo que cambia son los ángulos del parche.

¿Qué se hace con las salidas de los 16 parches?

Se reúnen esos 16 vectores de 4 canales y lo rearmamos en su lugar formando un mapa de características de tamaño (4 canales, 4×4).

¿Se promedian al final?

Recordemos que tenemos dos opciones en la cabeza clásica:

GAP: Se promedia cada canal sobre su malla 4×4, lleva a que quedan 4 números (uno por canal) y luego una Linear(4→1). Es compacto, pero puede perder detalle.

Flatten: no se promedia, se concatenan los 64 valores (4 canales × 4×4) a Linear(64→32) a ReLU a Dropout a Linear(1). Más capacidad para aprender.

¿Cada imagen conserva sus píxeles?

Sí. Nunca se mezclan pixeles de una imagen con otras imágenes. Cada imagen se trocea en sus **propios** 16 parches y cada parche genera su vector de 4 salidas.

¿Cómo aprende el ansatz si es el mismo para todos?

En el entrenamiento, se calcula la pérdida sobre el batch; PyTorch/PennyLane hacen backprop a través de todas las llamadas del QNode (los 16 parches por imagen), agregando la información de gradiente. Así, los mismos parámetros de theta se ajustan para funcionar bien en todas las posiciones.