Entrenamiento de operadores cuánticos

Arquitectura y programación de computadores cuánticos

Curso 2023–2024

David Peromingo Peromingo

Profesor
Guillermo Botella Juan

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Resumen

Entrenamiento de operadores cuánticos

El campo del aprendizaje automático ha experimentado un avance sin precedentes en los últimos años, extendiéndose a áreas como la construcción de circuitos cuánticos. Herramientas como PennyLane han traído consigo nuevas formas para desarrollar esta tendencia, y llevar a un público más amplio esta fusión de computación cuántica y aprendizaje automático.

En este trabajo veremos cómo implementar y entrenar circuitos cuánticos con PennyLane y PyTorch, orientándolos a la construcción de operadores aritméticos. Con este enfoque se busca alejarse de los muchos ejemplos de circuitos cuánticos de clasificación y regresión que encontramos comúnmente en tutoriales y demos, que buscan imitar redes neuronales clásicas en problemas triviales. Para ello, construiremos primero un circuito simple con el que comprender el proceso de entrenamiento, para posteriormente pasar a un operador más elaborado y profundo con el que poner a prueba estas técnicas. También analizaremos las posibilidades que nos brinda esta forma automática de construir y optimizar circuitos, frente a realizarlos manualmente.

Índice

Capítulo 1

Introducción a PennyLane y PyTorch

1.1. Entorno de PennyLane

El framework de PennyLane [1] es un kit de desarrollo de software de código abierto que integra bibliotecas de aprendizaje automático con simuladores cuánticos y *backends*, dando a los usuarios la capacidad de diseñar y entrenar circuitos cuánticos.

Un programa en PennyLane se organiza comúnmente en nodos, que pueden ser clásicos (conteniendo código clásico) o cuánticos, y forman redes análogas a las neuronales. Por simplicidad, en este trabajo sólo trataremos redes de un único nodo cuántico. Un nodo cuántico o *QNode* encapsula la función que inicializa el circuito cuántico y el dispositivo específico que lo ejecuta, y puede interactuar con bibliotecas de aprendizaje automático como PyTorch [2] para entrenar partes parametrizables del circuito.

A la hora de construir un circuito en uno de estos *QNodes*, PennyLane nos proporciona muchas utilidades. Sea el siguiente ejemplo de código de *QNode*:

```
import pennylane as qml
n_qubits = 4
dev = qml.device("default.qubit", wires=n_qubits)

@qml.qnode(dev)
def circuit(inputs, weights):
  qml.AngleEmbedding(inputs, wires=range(n_qubits))
  qml.Hadamard(wires=range(n_qubits))
  return [qml.expval(qml.PauliZ(wires=i)) for i in range(n_qubits)]
```

El primer paso es inicializar el dispositivo que ejecutará el circuito de 4 qubits, eligiendo uno entre los disponibles en el entorno (el predeterminado es "*default.qubit*"). El nodo comienza con una directiva que asocia el dispositivo *dev* con la función de inicialización inmediatamente posterior, en la que se definen todos los elementos que

contendrá el circuito. Cada uno de ellos se integra en el circuito automáticamente llamando a sus respectivos constructores, y asignando la lista de índices de los qubits o *wires* sobre los que actúan.

La función del circuito puede tener un número arbitrario de argumentos, y todos ellos a excepción de aquel que tenga el nombre *inputs* se consideran parámetros entrenables. Estos parámetros son los que utilizaremos para el aprendizaje de nuestros modelos, mientras que *inputs* sirve para definir las entradas del circuito mediante capas como *AngleEmbedding*. Esta última se limita a realizar una rotación en cada qubit (de forma predeterminada en el eje X de la esfera de Bloch) en función de los valores de esta lista de entradas, de modo que podemos inicializar cada qubit a 0 o 1 mediante rotaciones de 0 y π radianes.

La salida del nodo debe ser el resultado de una medida del circuito. En nuestro caso utilizaremos la función $expval(m)$, que devuelve el valor esperado al aplicar la medida m. El valor esperado puede ser uno de los autovalores de m (0 y 1 en el caso de la base computacional) o un valor intermedio proporcional a su probabilidad si el estado está en superposición. No obstante, no es posible utilizar este $expval(m)$ con m midiendo la base computacional, puesto que no tenemos una matriz/puerta con autovalores 0 y 1 y autovectores $|0\rangle$ y $|1\rangle$. Nos interesa usar la matriz de Pauli Z en su lugar, con autovectores $|0\rangle$ y $|1\rangle$ y autovalores 1 y -1 respectivamente, de modo que una medida con valor esperado $v \in [-1, 1]$ se traduce a una probabilidad $p = \frac{-v+1}{2}$ de obtener el estado $|0\rangle$, y $1 - p$ de obtener el estado $|1\rangle$.

Conociendo la construcción de circuitos con Qiskit [3], el resto de recursos que utilizaremos son análogos.

1.2. Integración de PyTorch

PyTorch es una librería de aprendizaje automático de código abierto, muy extendida tanto en ámbitos empresariales como académicos y didácticos. Proporciona una interfaz sencilla para implementar y entrenar todo tipo de modelos de inteligencia artificial, y cuenta con su propio *plug-in* en PennyLane.

Los modelos de Deep Learning de PyTorch trabajan con tensores, arrays multidimensionales con la capacidad de interpretarse como parámetros entrenables. Así, los tensores que contengan valores variables de la red pueden modificarse progresivamente durante el entrenamiento, mediante descenso del gradiente, para que converjan a valores que proporcionen una salida más próxima a la solución de un problema.

Las redes de estos modelos se organizan en capas conectadas entre sí, cada una con sus correspondientes parámetros. Para interpretar un circuito cuántico como una capa de PyTorch, PennyLane nos proporciona el constructor TorchLayer, asignando un tensor de dimensiones dadas a los parámetros de su $QNode$. Dado que trabajaremos con un único nodo, el modelo equivale a una única capa de este tipo. El siguiente código muestra un ejemplo de su uso, donde *weights* representa las dimensiones del tensor asociado a la capa (en este caso 2×10, equivalente a una matriz):

```
capaCuantica = qml.qnn.TorchLayer(circuit, {"weights":(2,10)})
```

Para entrenar un modelo necesitaremos una función de pérdida a minimizar para derivarla y obtener los gradientes, y PyTorch nos ofrece una variedad de ellas. Una de las más básicas es L1, que mide el error absoluto medio [4] (MAE) entre la salida del modelo y el valor esperado como:

$$MAE(x, y) = \frac{1}{n} \sum_{i=0}^{n} |x_i - y_i| \tag{1.1}$$

Esta función de pérdida se utiliza junto con un optimizador, un objeto encargado de actualizar los parámetros en función de los gradientes calculados. El más básico, SGD (*Stochastic Gradient Descent* [5]), implementa directamente el descenso del gradiente.

Capítulo 2

Entrenamiento de un operador

Vamos a comenzar este pequeño proyecto entrenando un operador sencillo y cuya implementación conozcamos, para asegurar que el circuito obtenido como resultado del entrenamiento sea correcto. Un buen punto de partida es el sumador Draper visto en la práctica 3 de la asignatura. Este operador tiene una parte fácilmente parametrizable: las rotaciones de la suma entre las QFTs. Así, es suficiente con especificar el *layout* del circuito, y será el proceso de entrenamiento el que determine los valores de las rotaciones, que idealmente acabarán convergiendo a los valores que conocemos.

2.1. Especificación del circuito

Recordando el circuito del sumador Draper no aproximado, este comienza con una QFT sobre los qubits del primer operando y termina con la QFT inversa. En medio se realizan una serie de rotaciones tal como se presentan en la figura 2.1, donde el primer cable corresponde al qubit menos significativo.

PennyLane, por el contrario, interpreta el primer qubit como el más significativo, por lo que estas rotaciones se implementarían en código como:

```
def rotaciones(weights):
  count = 0
  for i in reversed(range(n_qubits//2)):
    for j in reversed(range(i+1)):
        qml.CRZ(weights[count], [n_qubits-j-1, i])
        count += 1
```

Lo que lleva al siguiente *QNode* del sumador:

```
@qml.qnode(dev)
def circuit(inputs, weights):
```

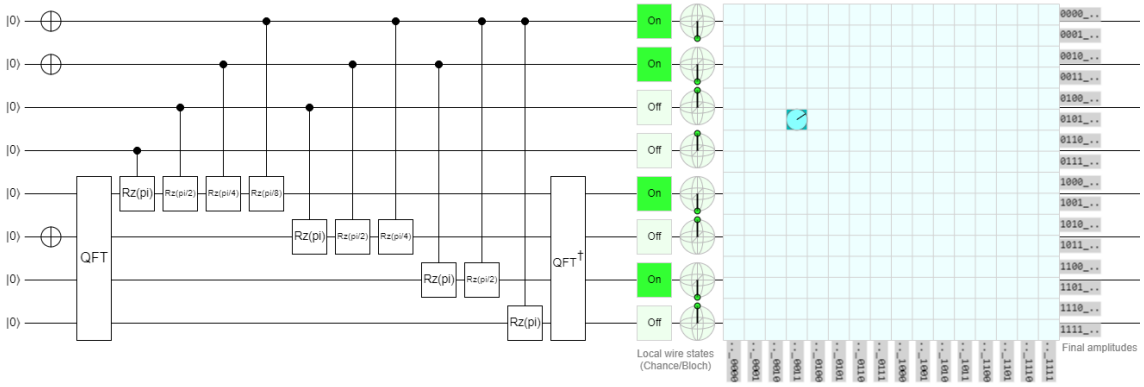

Figura 2.1: Diagrama del sumador en Quirk. Las puertas X al inicio del circuito representan un ejemplo de operandos de entrada (2,3), con el resultado de la suma marcado en la tabla de la salida (5,3). Aquí, el primer operando (el inferior en este orden) es el que almacena el resultado, y se mantiene el valor del segundo para asegurar la reversibilidad.

```
inputs *= math.pi
qml.AngleEmbedding(inputs, wires=range(n_qubits))
qml.QFT(wires=range(n_qubits//2))
rotaciones(weights)
qml.adjoint(qml.QFT(wires=range(n_qubits//2)))
return [qml.expval(qml.PauliZ(wires=i)) for i in range(n_qubits)]
```

Inicialmente, los valores asignados a las rotaciones (*weights*) estarán inicializados a valores aleatorios. Con esto, el circuito está preparado para utilizarse como TorchLayer en un entrenamiento.

```
sumador = qml.qnn.TorchLayer(circuit, {"weights":(10)})
```

Recordemos también que los ángulos de rotación correctos que deberá aprender el modelo con 4 qubits por operando son, siguiendo el orden de la figura 2.1, $\{\pi, \pi/2, \pi/4, \pi/8, \pi, \pi/2, \pi/4, \pi, \pi/2, \pi\}$.

2.2. Conjunto de datos

Durante el entrenamiento, el modelo necesitará tanto los operandos de entrada como la salida esperada para dichos operandos. Estos se representarán como una tupla de listas binarias de la forma:

$$(x, y) = (binario(op1) + + binario(op2), binario(op1 + op2) + + binario(op2)), \quad (2.1)$$

donde x es la entrada, y la salida esperada y '++' la concatenación de listas. Además, para evitar desbordamiento utilizando 4 qubits por operando, incluiremos

en este conjunto de datos solamente las sumas cuyo resultado sea menor que 16. Esto nos deja con un total de 136 sumas posibles.

2.3. Entrenamiento

En cada iteración del proceso de entrenamiento, se obtiene la salida del modelo dados dos operandos del conjunto de datos anterior. Esta salida, traducida a una lista de números entre 0 y 1 (recordemos que los autovalores esperados de las medidas están originalmente entre -1 y 1), se utiliza junto con la salida esperada en la función de pérdida. Por último, la pérdida resultante y el optimizador ajustan los parámetros del modelo. El código que permite este proceso es fundamentalmente el siguiente:

```
def train(model, train_set, epochs=15, lr=0.5):
  opt = torch.optim.SGD(model.parameters(), lr=lr)
  loss = torch.nn.L1Loss()
  avg_loss_list = []
  for epoch in range(epochs):
    running_loss = 0
    for (x,y) in train_set:
        opt.zero_grad()
        x = torch.Tensor(x)
        y = torch.Tensor(y)
        y_pred = formatoSalida(model(x))
        loss_evaluated = loss(y_pred, y)
        loss_evaluated.backward()
        opt.step()
        running_loss += loss_evaluated
```

El *lr* o *learning rate* es el coeficiente que indica al optimizador el ritmo de aprendizaje que debe llevar el entrenamiento, por lo que valores altos lo aproximarán más rápidamente a la solución óptima, a costa de perder un poco de precisión. El resultado de una sesión de entrenamiento de 10 épocas (iteraciones) y $lr = 0{,}5$ se muestra en la figura 2.2 como una gráfica de la pérdida media por época.

Este entrenamiento duró escasos segundos en un entorno de Google Colab, y su consumo de memoria también fue irrisorio.

2.4. Resultados

Llamamos exactitud o *accuracy* al porcentaje de resultados correctos que arroja el modelo para un conjunto de entradas. Dado que los resultados arrojados por el modelo son salidas esperadas entre 0 y 1, podemos interpretarlas como parcialmente correctas en función de su diferencia con el resultado real (e.g. una salida $y = 0{,}71$ será correcta en un 71 % si la salida real es 1, y en un 29 % en otro caso).

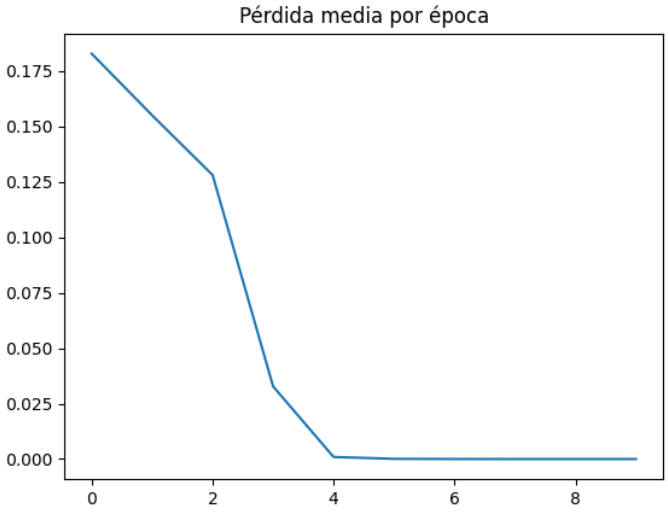

Figura 2.2: Representación gráfica de la pérdida media obtenida por MAE en cada época. Podemos observar que en tan sólo 5 de las 10 épocas alcanza un estado prácticamente óptimo.

El sumador final presenta un *accuracy* del 99.9995 % en tan solo 10 épocas, y los parámetros se ajustan a la perfección a los ángulos de rotación esperados. El tensor final de parámetros, dividido por π para distinguir sus coeficientes más fácilmente, es el siguiente:

```
Parameters/pi: tensor([0.9930, 0.5000, 2.2478, 0.1283, 1.0000, 0.5000,
    2.2500, 1.0000, 0.5000, 1.0000], grad_fn=<DivBackward0>)
```

Prácticamente idénticos a los correctos tomando módulo 2 ($2\pi \equiv 0$).

En el código del proyecto se pueden ejecutar ejemplos de sumas correctas, con su representación binaria y decimal; incluso para sumas que presentan desbordamiento (estas últimas *mod* 16), para los cuales el modelo acierta pese a no haberlas aprendido directamente.

2.5. Extra: Restador

Adicionalmente, podemos reutilizar el circuito anterior para crear un operador nuevo: un restador. Intuitivamente, tiene sentido que equivalga a un sumador con rotaciones en sentido contrario, pero vamos a descubrirlo siguiendo el mismo procedimiento.

La única modificación que debemos hacer es que el conjunto de datos tenga como salidas esperadas la resta de sus operandos, en lugar de la suma:

$$(x, y) = (binario(op1) + + binario(op2), binario(op1 - op2) + + binario(op2)). \quad (2.2)$$

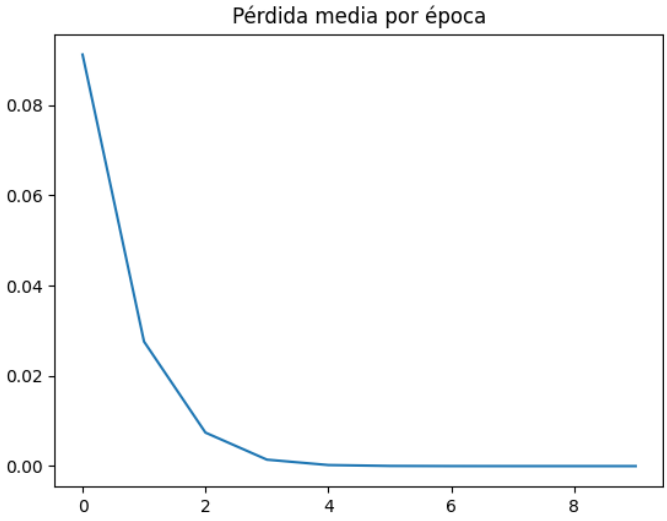

Figura 2.3: .

Repitiendo el bucle de entrenamiento, obtenemos resultados prácticamente idénticos, tanto en pérdida como en *accuracy*. También podemos comprobar que, efectivamente, los parámetros obtenidos son los esperados o equivalentes.

```
Parameters/pi: tensor([ 1.0003, -0.4999, 1.7499, 1.8748, 1.0000,
    1.5000, -0.2500, 1.0000, -0.5000, 1.0000 ], grad_fn=<DivBackward0>)
```

Capítulo 3

Entrenando circuitos complejos: multiplicador

Después de entrenar exitosamente un circuito sencillo mediante técnicas de aprendizaje automático, es el momento de llevar el proceso a un nivel superior con un circuito más complejo.

Este será un multiplicador [6], el cual no hemos visto en la asignatura. Conceptualmente no es más complicado que el sumador, pero la profundidad del circuito que lo implementa y el número de rotaciones necesarias es sensiblemente mayor. Para entrenarlo seguiremos un procedimiento similar al del sumador, y comprobaremos la capacidad de aprendizaje de estos modelos cuando se enfrentan a circuitos cuánticos grandes.

3.1. Especificación del circuito

Para multiplicar manualmente dos números de n cifras a y b, la operación que conocemos es multiplicar cada dígito $a_n a_{n-1} \cdots a_1$ por b, y sumar los resultados con un desplazamiento a la izquierda de i posiciones para el i-ésimo dígito. Como trabajamos en binario, multiplicar $a_i \cdot b$ resulta en 0 si $a_i = 0$ y b en otro caso; mientras que un desplazamiento a la izquierda de i posiciones equivale a multiplicar por $2i$.

Traduciéndolo a un circuito, podemos agregar un tercer operando para almacenar la solución inicializado a 0, al que $\forall i : 0 < i \leq n$ se suma b $2i$ veces si $a_i = 1$. Si reutilizamos el sumador de la sección anterior y le incorporamos como bit de control dicho a_i, este circuito multiplicador tiene el aspecto de la figura 3.1.

Dado que conocemos los valores correctos de las rotaciones de cada sumador, podremos verificar los parámetros aprendidos por el modelo. Este circuito suma un total de 150 parámetros entrenables divididos entre los 15 sumadores que lo componen, un número mucho mayor a los 10 que tenía un único sumador.

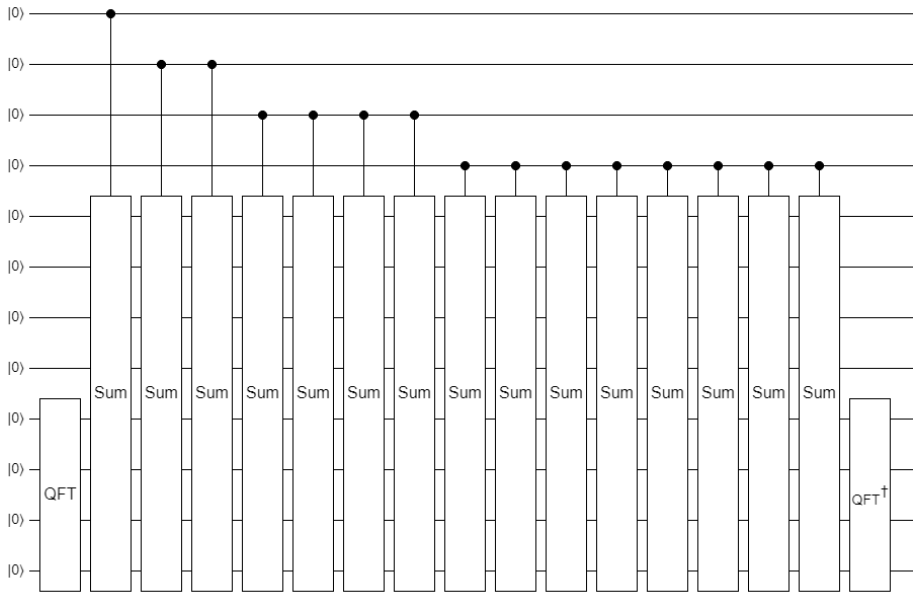

Figura 3.1: Diagrama del multiplicador en Quirk para 4 qubits por operando.

El siguiente código implementa el *QNode* correspondiente:

```
n_qubits = 12
n_qubits_op = 4  #qubits por operando
wires = range(n_qubits)   #lista de cables, uno por qubit
w_op1 = range(n_qubits_op)
w_op2 = range(n_qubits_op, 2*n_qubits_op)
w_res = range(2*n_qubits_op, n_qubits)

@qml.qnode(dev)
def circuit(inputs, weights):
  inputs *= math.pi
  qml.AngleEmbedding(inputs, wires=range(n_qubits))
  count = 0
  qml.QFT(wires=w_res)
  for i in range(n_qubits_op):
    for j in range(2**i):
      qml.ctrl(sumador, [w_op2[i]])(weights[count], w_res, w_op1)
      count += 1
  qml.adjoint(qml.QFT(wires=w_res))
  return [qml.expval(qml.PauliZ(wires=i)) for i in range(n_qubits)]


multiplicador = qml.qnn.TorchLayer(circuit, {"weights":(15,n_rotaciones)})
```

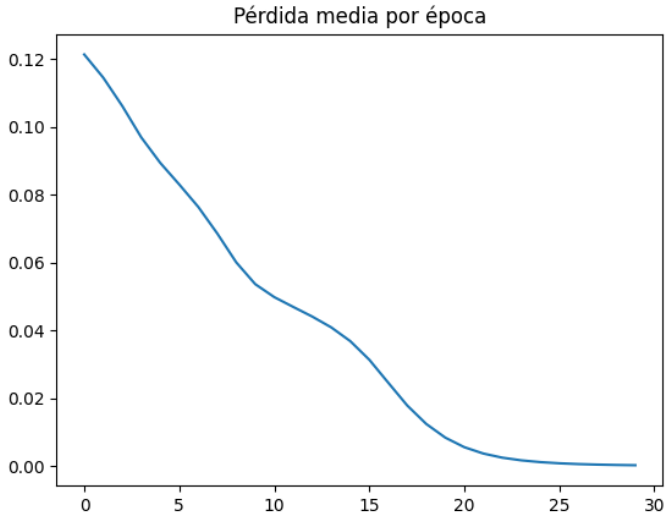

Figura 3.2: Pérdida media por época durante el entrenamiento del multiplicador.

3.2. Conjunto de datos

El conjunto de datos de entrenamiento seguirá el mismo formato que el utilizado para el sumador, añadiendo un tercer operando para la solución.

Asimismo, mantenemos la restricción de utilizar operandos cuyo producto no produzca desbordamiento, resultando en un conjunto de 60 multiplicaciones. Aquellos operandos que no cumplan esta restricción pueden utilizarse para validar el circuito una vez entrenado, verificando así que es capaz de generalizar con valores cuyo resultado no ha aprendido directamente.

3.3. Entrenamiento

Partiendo del bucle de entrenamiento del sumador, conviene como mínimo aumentar el número de épocas de entrenamiento. Asumimos también que el tiempo que tomará cada época será notablemente mayor, así como el coste en memoria. Así, un entrenamiento de 30 épocas asciende a 12 minutos, consumiendo hasta 4GB de memoria RAM en el mismo entorno de Google Colab. Las métricas de este entrenamiento se muestran en la figura 3.2.

3.4. Resultados

El multiplicador final presenta un accuracy del 99.98 %. Curiosamente y a diferencia del sumador, los valores finales de los parámetros entrenados difieren claramente de los esperados, presentando además discrepancias entre las mismas rotaciones de dos sumadores distintos del circuito. Sin embargo, este hecho no afecta negativamente a su desempeño, por lo que esta nueva configuración es en realidad una nueva

solución correcta para las rotaciones del multiplicador. Estos parámetros finales son:

```
Parameters/pi: tensor([
        [ 0.2412,  0.2500,  0.6593,  1.0525,  1.2834,  1.3438,  0.0037,  1.3705,
          1.9543,  1.9816],
        [ 1.7799,  0.1285,  1.2978,  0.5640,  0.7471,  2.2109,  1.6605,  1.3002,
          0.6050,  0.0384],
        [ 1.6786,  1.8702,  1.7018, -0.0804,  0.8176,  1.7862,  1.3403,  0.6863,
         -0.5846, -0.0334],
        [ 0.6861,  1.8698,  1.1321,  1.5230,  1.6596,  1.5269,  1.6141,  0.0398,
          0.4859,  0.3868],
        [ 0.0867,  1.4617,  1.4280,  1.4356,  1.3062,  1.1607,  1.5447,  0.1615,
          1.1477,  1.5175],
        [ 0.7162,  2.2001,  1.3218,  1.5727,  1.7278,  0.0197,  0.0585,  0.6526,
          1.6881,  0.4288],
        [ 0.6070,  1.4646,  0.6310,  1.7081,  1.3070,  0.2946,  1.2805,  1.1428,
          1.6816,  1.6691],
        [ 1.5892,  0.4447,  0.7416,  1.7257,  0.2816,  0.2602,  1.3245,  0.4982,
          0.1029,  1.0389],
        [ 0.9833, -0.0275,  1.7800,  1.6014,  1.2656,  1.8374,  0.8455,  1.8293,
          0.3771,  1.7109],
        [ 0.2411,  0.9568,  1.5368,  1.2830,  0.6640,  0.9832,  1.1880,  1.1059,
          1.4280,  1.3692],
        [ 1.3256,  0.7180,  0.3991,  0.5180,  1.8095,  0.8949,  1.0363,  0.6525,
          0.9265,  1.1111],
        [ 1.7680,  0.7220,  1.8908,  1.8974,  0.1021,  0.5825,  0.7588,  1.3760,
          0.1454,  1.9592],
        [ 1.0987,  0.6475,  0.8109,  0.8692,  1.4065,  1.9639,  1.3808,  2.0591,
          0.1773,  0.9566],
        [ 0.9346,  0.4518,  0.2500,  0.2445,  0.6294,  1.3967,  1.1384,  0.9223,
          0.7452,  0.2057],
        [ 1.0565,  0.5893,  0.8437, -0.0165,  0.8410,  0.5818,  0.5775,  0.5554,
          0.5987,  0.6482]], grad_fn=<DivBackward0>)
```

Cada fila de la matriz de parámetros anterior representa los ángulos de rotaciones de cada sumador. Nuevos tensores de parámetros pueden obtenerse ejecutando de nuevo el entrenamiento. En el próximo capítulo exploraremos las posibilidades que trae esta discrepancia con los parámetros originales del multiplicador.

Capítulo 4

Optimización de circuitos

Los resultados anteriores nos muestran un comportamiento interesante de los parámetros del circuito. Cuando las rotaciones en uno de los sumadores son mayores o menores de las que le corresponden, los parámetros de los demás aprenden a compensar este desfase, llegando a una solución óptima con rotaciones diferentes de las esperadas inicialmente.

Aquí nos asalta una pregunta, ¿Sería capaz el circuito de compensar un sumador con todas sus rotaciones a 0?. Dicho de otro modo, ¿Podría aprender a mantener su desempeño eliminando uno de los sumadores? Si así fuera, este ajuste de parámetros podría llevarnos a circuitos menos profundos, y por consiguiente más eficientes, sin costes adicionales.

4.1. Eliminación de un sumador

Para obtener la respuesta a la pregunta anterior, vamos a utilizar el mismo circuito del multiplicador eliminando el último sumador, como indica la figura 4.1.

Sin más demora, vamos a pasar directamente al entrenamiento y resultados. Pasando de 150 a 140 parámetros, el consumo de memoria y tiempo por época es prácticamente el mismo. Tras 60 épocas y 25 minutos, la conclusión del entrenamiento es clara: el circuito ha conseguido resultados perfectos con un sumador menos.

Con un accuracy de 99.998 % y el mismo ratio de productos correctos que el circuito sin optimizar, los parámetros han aprendido una configuración de valores que permiten reducir la profundidad en 8 puertas, la profundidad de un sumador. Este resultado demuestra que es posible optimizar circuitos grandes por este método, por lo que podemos llevarlo un paso más allá eliminando más de un sumador.

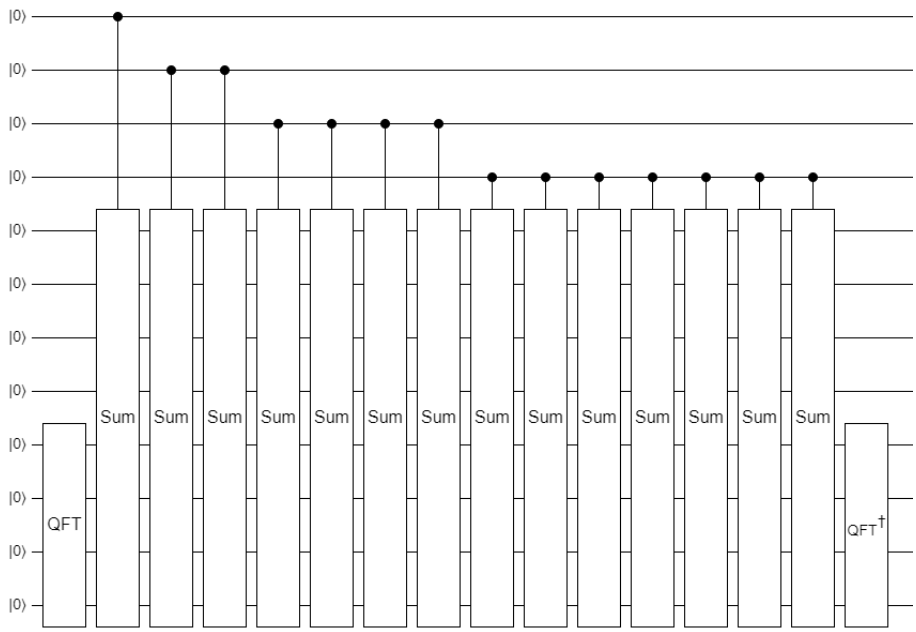

Figura 4.1: Diagrama del multiplicador en Quirk, al que se le ha extraído el último sumador.

4.2. Multiplicador con (casi) la mitad de profundidad

Tal como se estructura un multiplicador, su profundidad está en $\mathcal{O}(2^n)$, donde n es el número de qubits. En lugar de reducir este orden que complejidad, ya que puede afectar al funcionamiento del operador, vamos a intentar mantenerlo reduciendo la profundidad del circuito a la mitad sin contar la de las QFTs. Esta reducción consiste en eliminar la mitad de los sumadores condicionados por cada qubit del primer operando (a excepción del primer qubit, que sólo condiciona a uno), lo que resulta en un circuito como el de la figura 4.2.

Con un total de 80 parámetros, 100 épocas, casi 3GB de memoria y mismo coste de tiempo, los resultados son idénticos al anterior caso. Este entrenamiento, aplicado a circuitos de cada vez más qubits por operando, resultaría en una profundidad que tiende a la mitad de la del circuito sin optimizar.

Con esto se demuestra definitivamente que existe un impacto importante en la optimización que el aprendizaje automático puede proporcionar en circuitos cuánticos, y pruebas más exhaustivas pueden acotar hasta dónde esta optimización es capaz de llegar.

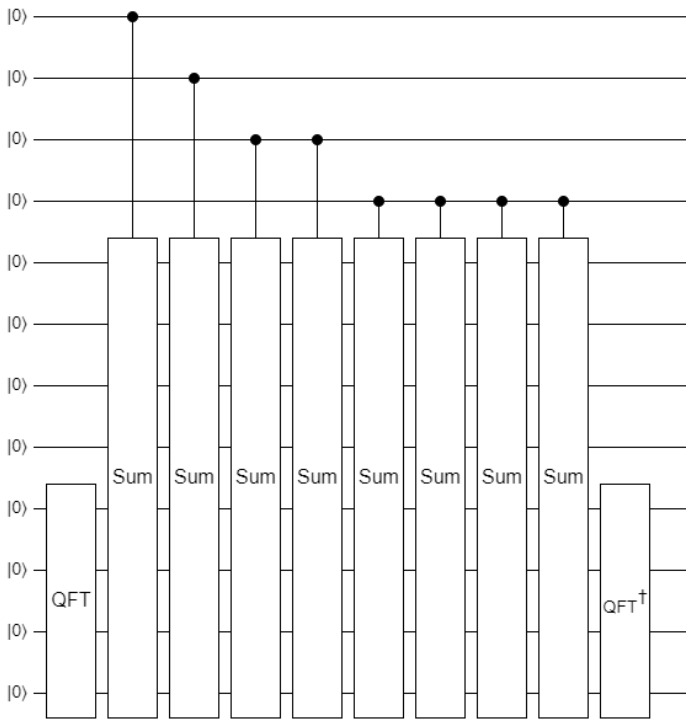

Figura 4.2: Diagrama del multiplicador en Quirk, al que se le han extraído la mitad de los sumadores condicionados a cada qubit, a excepción del primero.

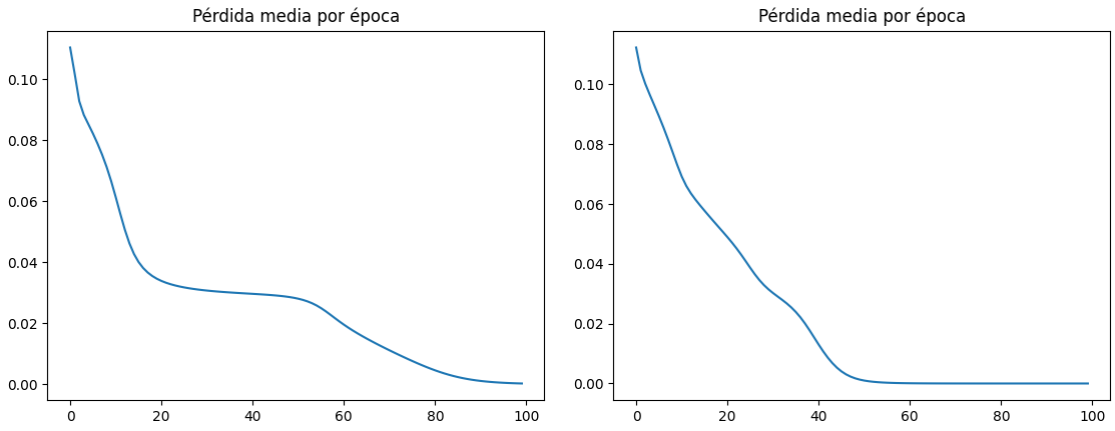

Figura 4.3: Métricas de los entrenamientos de multiplicadores con un sumador menos y la mitad de sumadores respectivamente. Se puede observar que, no solo no resulta más costoso entrenar un operador más optimizado, sino que el menor número de parámetros produce una convergencia más rápida hacia la solución óptima.

Capítulo 5

Conclusiones

A lo largo de este pequeño proyecto, hemos podido comprobar con casos prácticos la versatilidad de PennyLane tanto en construcción de circuitos como en integración en ellos de técnicas de aprendizaje automático. Al igual que estas, existen multitud de extensiones del framework y utilidades que convierten a PennyLane en un firme competidor de herramientas como Qiskit, superándolo en ciertos aspectos al tomar prestados recursos de otros frameworks como PyTorch.

Por otro lado, hemos aprendido a entrenar circuitos centrados en operaciones aritméticas, alejándonos de los típicos ejemplos de algoritmos de clasificación que encontramos en casi la totalidad de tutoriales de QML. Con esto ampliamos la mira de los circuitos entrenables mediante aprendizaje automático a todo tipo de operaciones, con la única limitación de necesitar una idea previa del layout del propio circuito.

Yendo un paso más allá, hemos comprobado que es posible obtener nuevas soluciones a circuitos que conocemos, como es el caso del multiplicador con rotaciones distintas de las que corresponden a sus sumadores internos. Además, estas nuevas soluciones se pueden extrapolar a versiones reducidas de dichos circuitos, en los que los parámetros aprenden a compensar esta reducción manteniendo el desempeño global del operador. El método propuesto es, por tanto, una forma de optimización de circuitos que permite reducirlos de formas que sólo las técnicas de inteligencia artificial son capaces de deducir.

En la historia reciente, para muchos problemas clásicos se han encontrado algoritmos mediante aprendizaje automático que rivalizan e incluso superan a los creados por humanos (es el caso de la multiplicación de matrices con *AlphaTensor* [7] de Google DeepMind). Es probable que, mediante estas mismas técnicas, el campo de la computación cuántica también se beneficie de nuevos y más eficientes circuitos que no podrían obtenerse por medios teóricos.

Bibliografía

[1] "Pennylane documentation." https://docs.pennylane.ai/en/stable/introduction/pennylane.html. Último acceso: 21 de mayo de 2024.

[2] "Pytorch documentation." https://pytorch.org/docs/stable/index.html. Último acceso: 18 de mayo de 2024.

[3] "Qiskit documentation." https://docs.quantum.ibm.com/. Último acceso: 21 de mayo de 2024.

[4] "Mean absolute error," in *Encyclopedia of Machine Learning and Data Mining* (C. Sammut and G. I. Webb, eds.), p. 806, Springer, 2017.

[5] S. Amari, "Backpropagation and stochastic gradient descent method," *Neurocomputing*, vol. 5, no. 3, pp. 185–196, 1993.

[6] "Tutorial: Multiplicando qubits." https://m.youtube.com/watch?v=rhwf-F0CnSc. Último acceso: 21 de mayo de 2024.

[7] "Discovering novel algorithms with alphatensor." https://deepmind.google/discover/blog/discovering-novel-algorithms-with-alphatensor/. Último acceso: 21 de mayo de 2024.