

# Proyecto Final APCC

Aplicación de TFQ a Codificación Superdensa  
y Teleportación Cuántica

Guillermo García Patiño Lenza

# Índice

- Repaso Codificación Superdensa y Teleportación Cuántica
- Circuitos parametrizados en Cirq
- Mediciones y Expectation Values
- Simulación con Tensorflow Quantum
- Construcción y entrenamiento de redes neuronales
- Conclusión

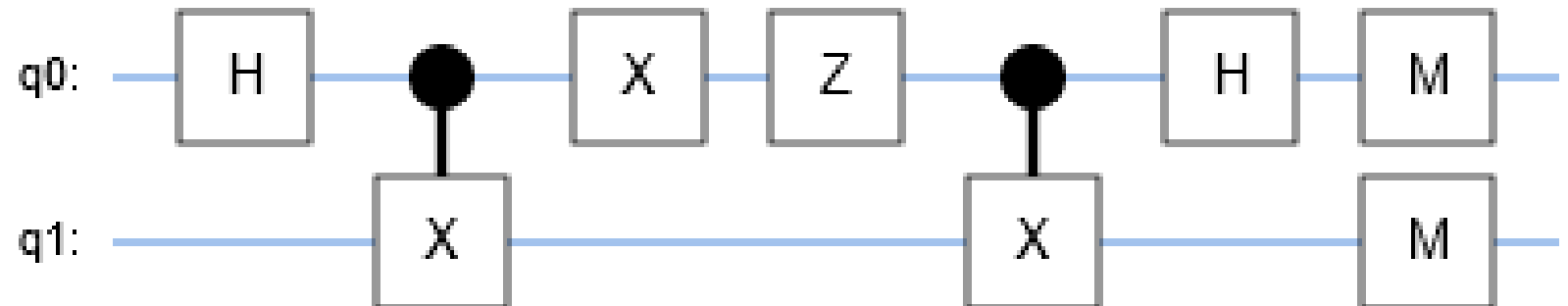


# Repaso de Codificación Superdensa y Teleportación Cuántica

# Codificación Superdensa

- Conseguir transmitir 2 bits clásicos enviando un único qubit

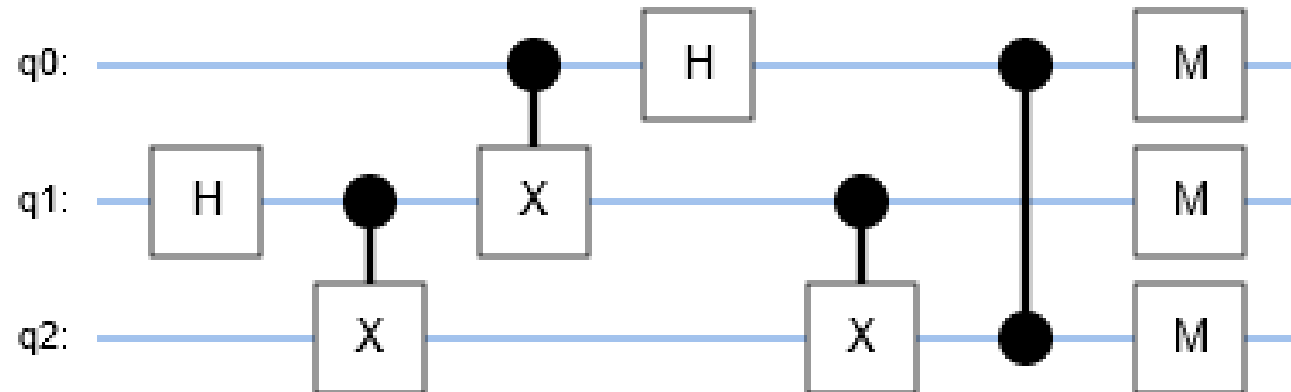
Si <i>Alice</i> quiere enviar	entonces <i>Alice</i> aplica el operador ...
00	Identidad
01	$X$
10	$Z$
11	$ZX$ (primero se aplica $X$ y luego $Z$ )



# Teleportación Cuántica

- Transmisión del estado de un qubit a otro qubit a una distancia

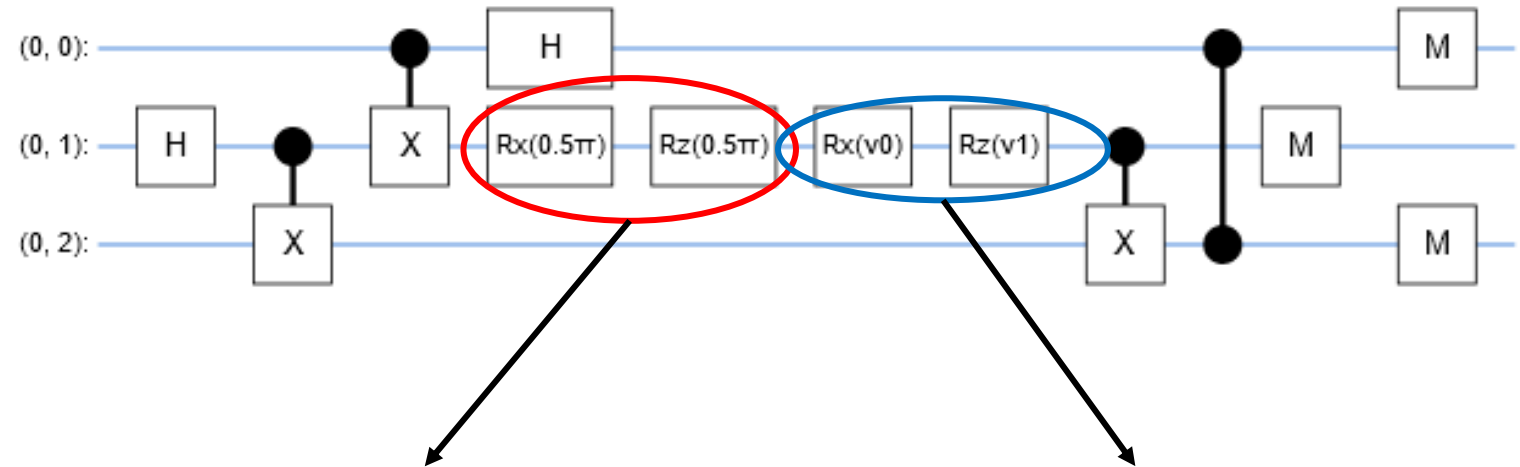
Si <i>Alice</i> transmite	entonces <i>Bob</i> aplica el operador ...
00	Ninguno (el qubit de <i>Bob</i> ya coincide con el de <i>Alice</i> )
01	$X$
10	$Z$
11	$ZX$ (primero se aplica $X$ y luego $Z$ )





# Circuitos Parametrizados en Cirq

# Puertas Parametrizadas en Cirq



Ruido parametrizado arbitrariamente

$V_0$  y  $V_1$  son los valores que el modelo aprenderá a calcular

## En el código

- Ruido parametrizado son rotaciones en torno al eje X ( bit flip error ) y en torno al eje Z ( phase flip error )

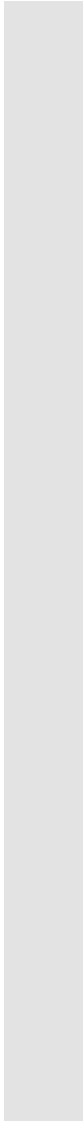

```
# Se inserta el ruido a la hora de enviar el bit
circuitoBase.append(cq.rx(gradoX).on(q1))
circuitoBase.append(cq.rz(gradoZ).on(q1))
```

- Se corrige con rotaciones en los mismos ejes.
- Estos parámetros son los que aprenderá la red neuronal de TFQ

```
ctr_params = sp.symbols('v0 , v1')
```

```
# Se insertan puertas X y Z para corregir el ruido
circuitoBase.append(cq.rx(params[0]).on(q1))
circuitoBase.append(cq.rz(params[1]).on(q1))
```





# Mediciones y Expectation Values

# Mediciones

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Diagram illustrating the eigenvalues and eigenvectors of the Pauli matrix  $\sigma_z$ .

The matrix is shown with its elements grouped into two columns. The first column contains 1 and 0, and the second column contains 0 and -1. The elements 1 and -1 are circled in red, and the elements 0 and 0 are circled in blue. Arrows point from the circled elements to the text "Autovalores" (Eigenvalues).

Below the matrix, the eigenvectors are shown:  $|0\rangle$  and  $-|1\rangle$ . Arrows point from the first and second columns of the matrix to these eigenvectors, which are then labeled "Autovectores" (Eigenvectors).

- Al medir empleando una de las puertas de Pauli, solo se puede obtener uno de los autovalores de la matriz.
- Cada autovalor está asociado a un autovector de la matriz de Pauli que se use para medir. Para medir en la base computacional, se usa la matriz Z de Pauli.

# Expectation Values

- Media de los valores obtenidos tras varias mediciones
- Para 1 qubit es un valor entre -1 y 1
- Se pueden combinar medidas libremente:
  - No medir todos los qubits
  - Multiplicar la medida de un qubit por un valor
- Ejemplo para 2 qubits:

```
# Operador para medir  
z2 = 2 * cq.Z(qubits[1]) + cq.Z(qubits[0])
```

q0 / q1	0	1
0	$2*1 + 1 = 3$	$2*(-1) + 1 = -1$
1	$2*1 - 1 = 1$	$2*(-1) - 1 = -3$

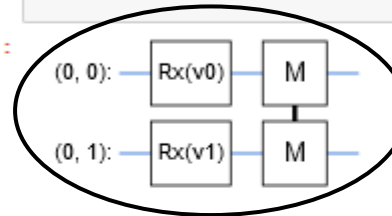


# Simulación con Tensorflow Quantum

# Simulación con Tensorflow Quantum

```
In [115]: qubit1 = cq.GridQubit.rect(1,2)
cparams = sp.symbols('v0 v1')
mom1 = cq.Moment([cq.rx(cparams[0]).on(qubit1[0]), cq.rx(cparams[1]).on(qubit1[1])])
mom2 = cq.measure(qubit1[0], qubit1[1])
caux = cq.Circuit([mom1, mom2])
SVGCircuit(caux)
```

Out[115]:



Circuito con 2 parámetros 'v0' y 'v1'

```
In [139]: valores1 = np.zeros((10,2))
valores2 = np.array(np.random.uniform(0, 2*np.pi, (10,2)), dtype = np.float32)
op = cq.Z(qubit1[0])

r1 = tfq.layers.Expectation()(caux, symbol_names=[cparams[0], cparams[1]], symbol_values=valores1, operators = op)
print(r1)
```

```
tf.Tensor(
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]], shape=(10, 1), dtype=float32)
```

Matrices (10,2)  
Son 10 pares de valores

Un resultado para cada par de valores

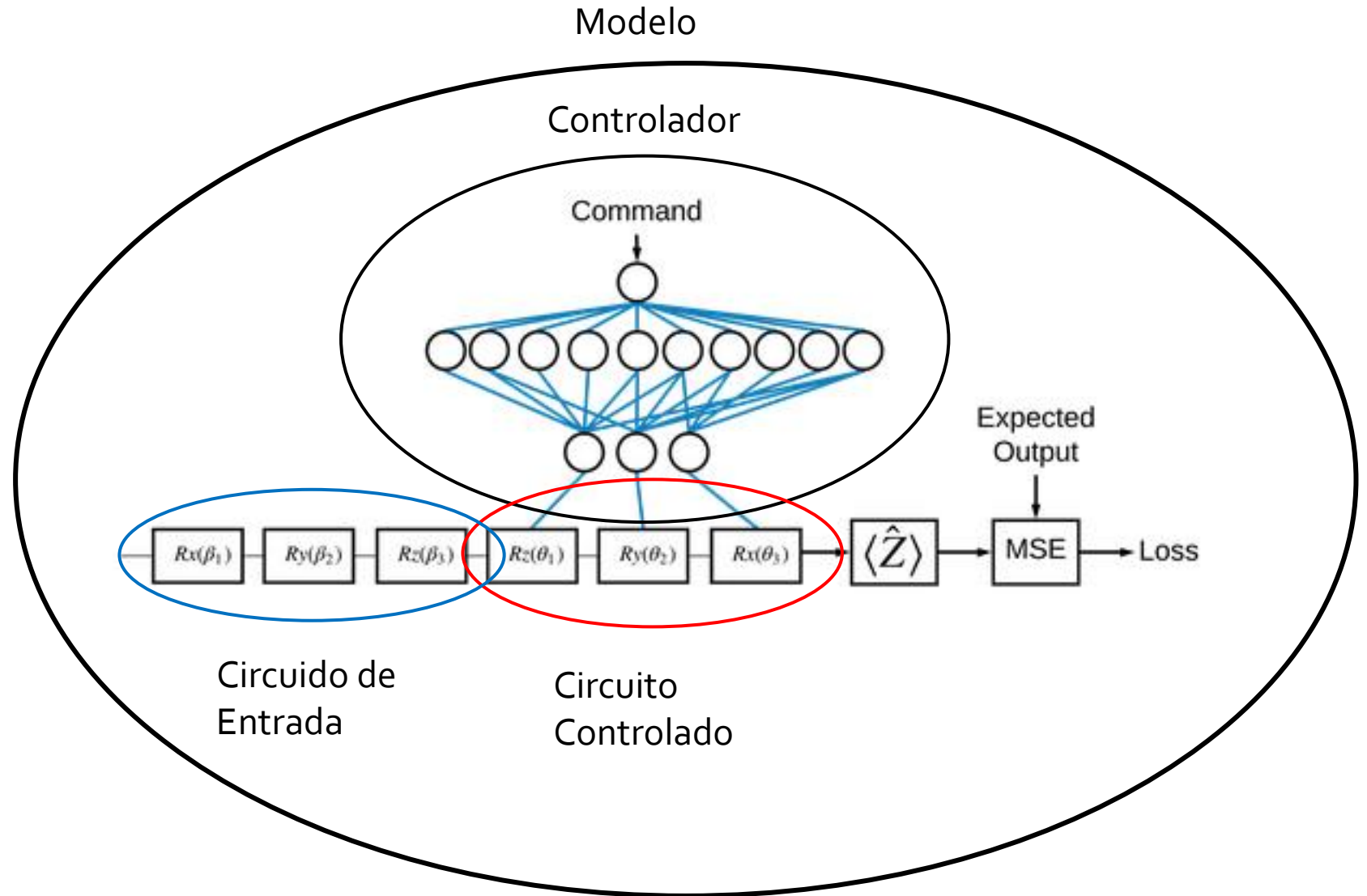
- Tensorflow Quantum da 'expectation values' como resultados'

# Construcción y Entrenamiento de Redes Neuronales

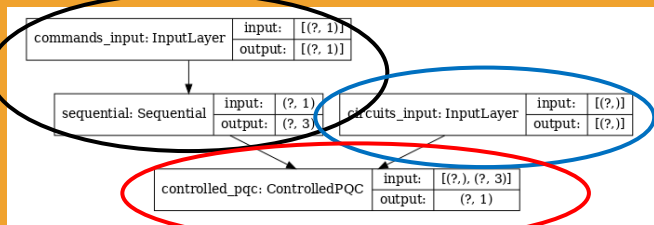
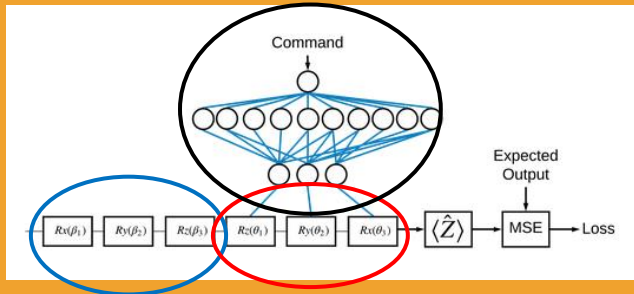
Ejemplo: en la codificación superdensa, aprender qué puertas tiene que aplicar Bob dependiendo de los bits que quiere transmitir Alice

# Modelo híbrido clásico - cuántico

- Controlador : recibe un comando y calcula los valores para producir la salida asignada al comando
- Circuito Controlado : produce una salida de acuerdo a los parámetros que asigna el controlador
- Circuito de Entrada : produce la entrada para el circuito controlado



# Modelo híbrido clásico-cuántico

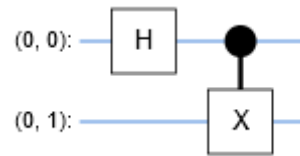


- Definición de entrada de comandos y de circuito :

```
ctr_params = sp.symbols('v0 , v1')
c, qubits, partes = producirCodifDensaGuess(ctr_params)

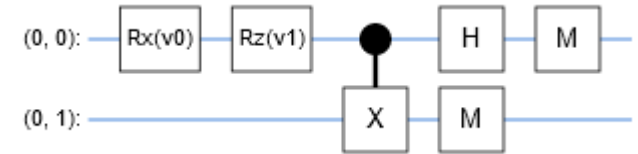
entrada = partes[0]
controlado = partes[1]
```

Entrada



```
# Configuración de las entradas
circuito_entrada = tf.keras.Input(shape = (), dtype = tf.string, name = 'circuito_entrada')
```

Controlado



```
# Operador para medir
z2 = 2*cq.Z(qubits[1]) + cq.Z(qubits[0])
```

```
# Configurar la capa de salida
capaExpected = tfq.layers.ControlledPQC(controlado, operators = z2)
```

- Construcción del controlador :

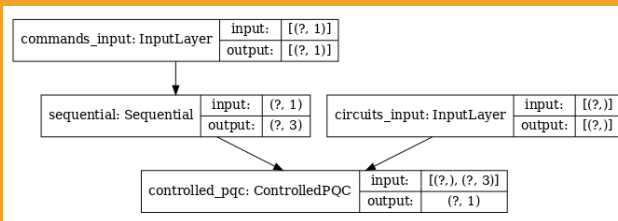
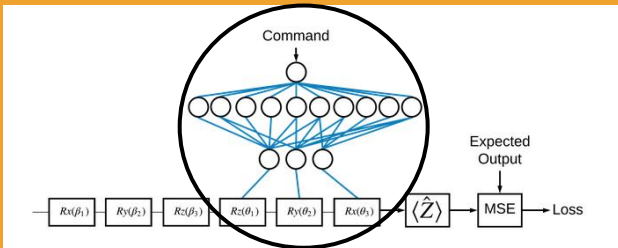
```
comandos_entrada = tf.keras.Input(shape = (1,), dtype = tf.dtypes.float32, name = 'entrada_comandos')
```

```
# Configuración del controlador
controller = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation = 'elu'),
    tf.keras.layers.Dense(2)
])
```

```
# Crear el controlador
capa1 = controller(comandos_entrada)
```



# Modelo híbrido clásico-cuántico

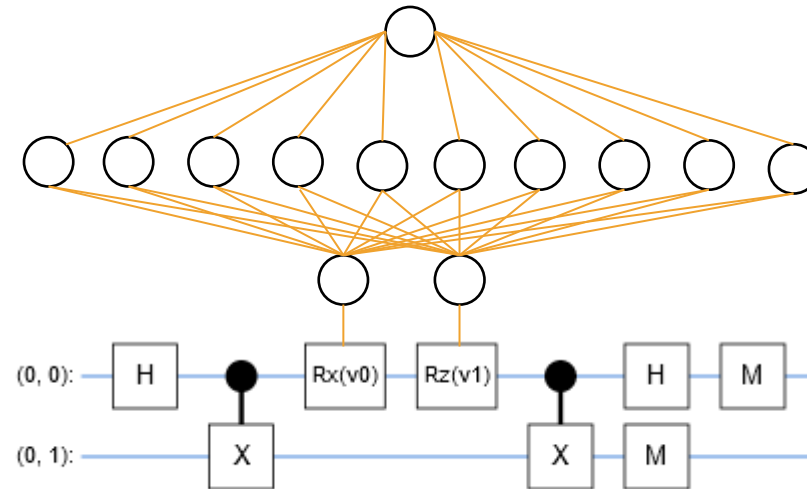


- Conectar Controlador a Circuito Controlado:

```
# Crear la capa de salida
calculo_expected = capaExpected([circuito_entrada, capa1])
```

- Definición del modelo

```
# Configurar el modelo
modelo = tf.keras.Model(inputs = [circuito_entrada, comandos_entrada], outputs = calculo_expected)
```



Buscamos obtener los parámetros  $v_0$  y  $v_1$  optimizados por el controlador

# Entrenamiento de la red neuronal

- Configuración de comandos y salidas esperadas:

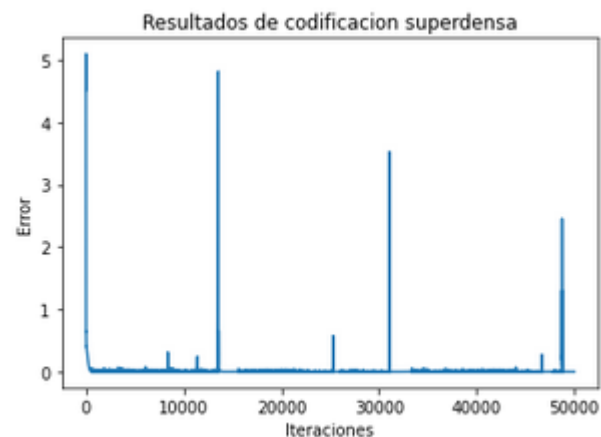
```
# Configurar Comandos y salidas para cada comando
comandos = np.array([[0],[1],[10],[11]], dtype = np.float32)
salidas_esperadas = np.array([[3],[1],[-1],[-3]], dtype = np.float32)
```

- Construir la entrada del circuito controlado:

```
# Generar un tensor con los 4 circuitos de entrada (una para cada entrada)
generador_datos = tfq.convert_to_tensor([entrada]*4)
```

- Entrenar el modelo:

```
# Entrenar el modelo
optimizer = tf.keras.optimizers.Adam(learning_rate = 0.05)
loss = tf.keras.losses.MeanSquaredError()
modelo.compile(optimizer = optimizer, loss = loss)
history = modelo.fit( x = [generador_datos, comandos], y = salidas_esperadas, epochs = 50000, verbose = 0)
```



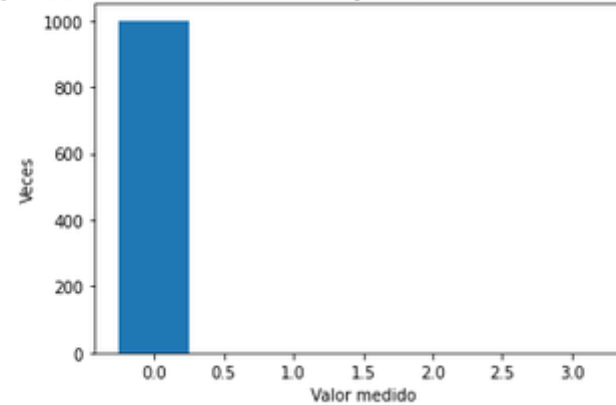
Para obtener los valores aprendidos para cada comando

```
ar = controller(tf.constant([[0],[1],[10],[11]])).numpy()
```

```
[[-1.23838186e-02  1.12730265e-02]
 [-5.03219783e-01 -8.70517540e+00]
 [ 2.50757837e+00 -1.97641029e+01]
 [ 3.07125711e+00 -2.18992691e+01]]
El error final es 2.138580020982772e-05
```

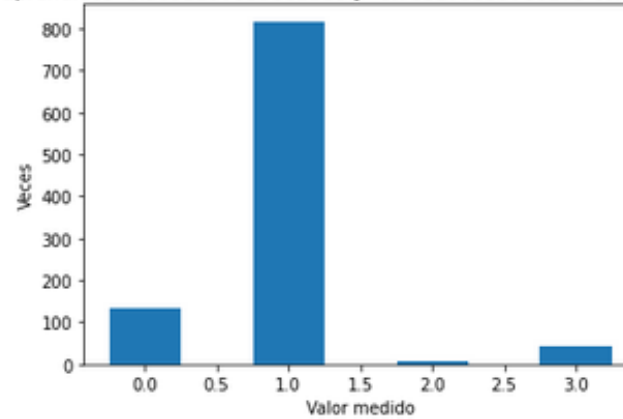
# Simulaciones con los parámetros resultado

Para los valores aprendidos -0.012383818626403809 y 0.011273026466369629 correspondientes al estado 0



$$\begin{aligned} V_0 &\sim 0 \\ V_1 &\sim 1 \end{aligned}$$

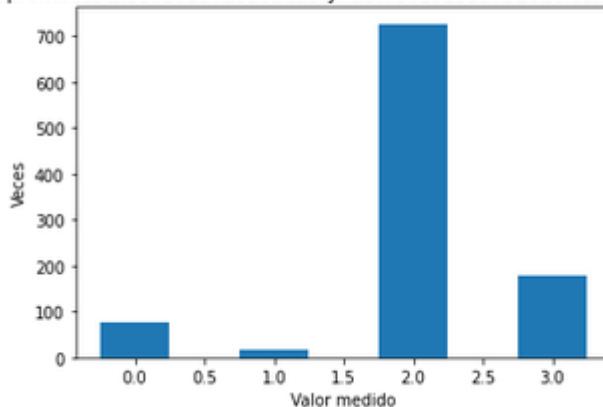
Para los valores aprendidos -0.5032197833061218 y -8.705175399780273 correspondientes al estado 1



$$\begin{aligned} V_0 &= -0.5 \\ V_1 &= -8.7 = -2.7 * \pi \end{aligned}$$

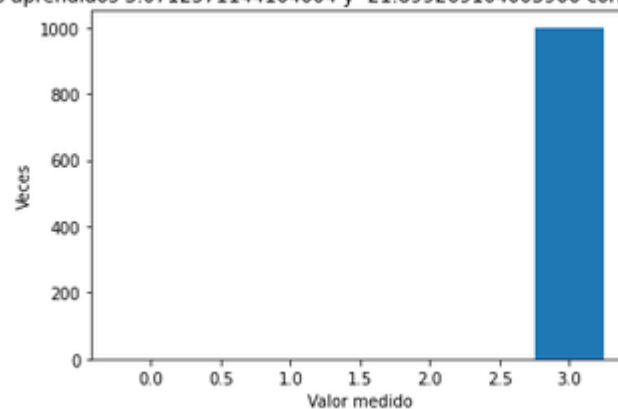
# Simulaciones con los parámetros resultado

Para los valores aprendidos 2.5075783729553223 y -19.764102935791016 correspondientes al estado 2



$$V_0 = 2.5$$
$$V_1 = -19.7 \sim -6.3 * \pi$$

Para los valores aprendidos 3.0712571144104004 y -21.899269104003906 correspondientes al estado 3

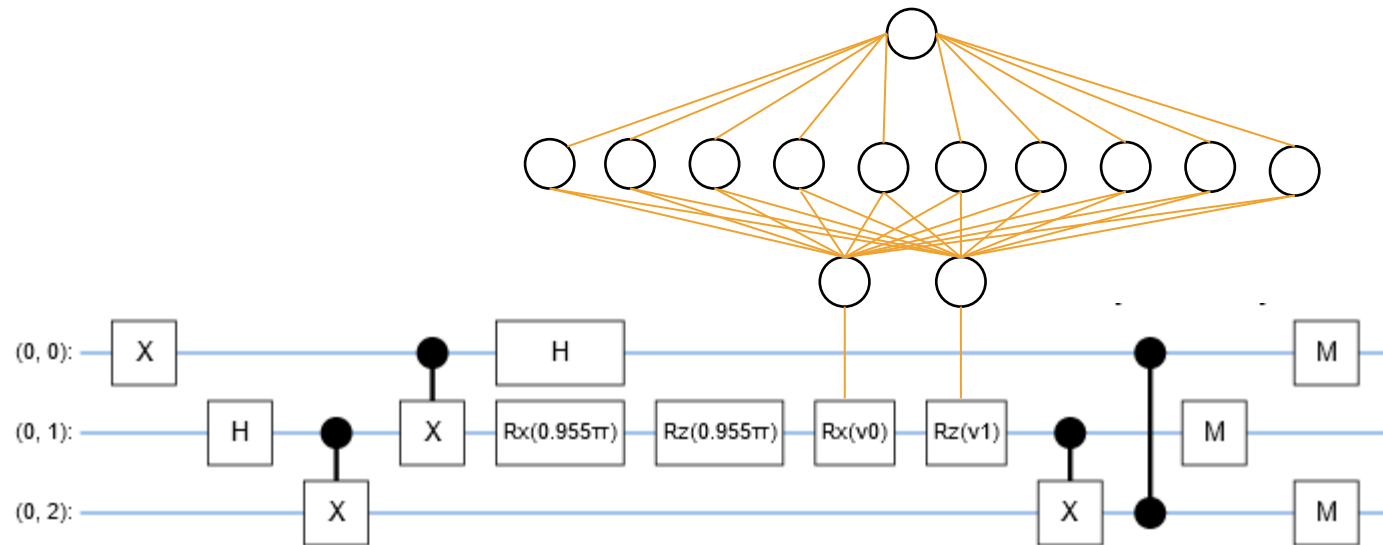


$$V_0 = 3.07 \sim \pi$$
$$V_1 = -21.9 \sim -7 * \pi$$

# Construcción y Entrenamiento de Redes Neuronales

Ejemplo: Aprender a corregir ruido generado con parámetros elegidos arbitrariamente

# Diferencias con el caso anterior



- En este caso, solo hay un comando posible, ya que se aplican por defecto las puertas que Alice necesita usar para transmitir un  $|1\rangle$

```
comandos = np.array([[1]], dtype = np.float32)
salidas_esperadas = np.array([[1]], dtype = np.float32)
```

- Como solo hay un comando disponible, el generador de datos solo usa una copia del circuito de entrada

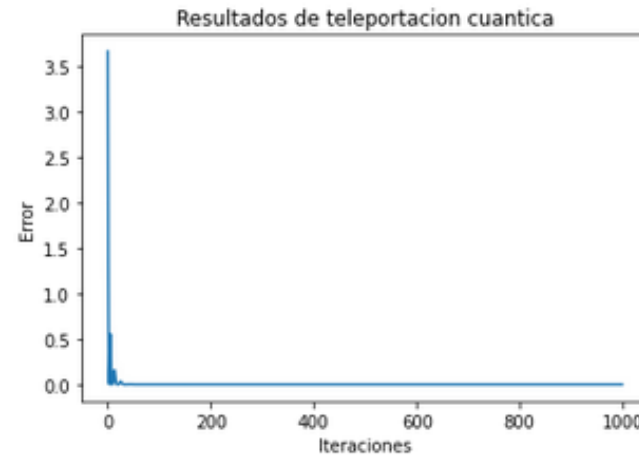
```
# Generar un tensor con los 2 circuitos de entrada (una para cada entrada)
generador_datos = tfq.convert_to_tensor([entrada])
```

- Solo se medirá el qubit de Bob, para ver si el estado del de Alice se ha teleportado correctamente

```
# Operador para medir
z2 = cq.Z(qubits[2])
```

# Resultados y Simulación con los parámetros resultado

- Resultados:



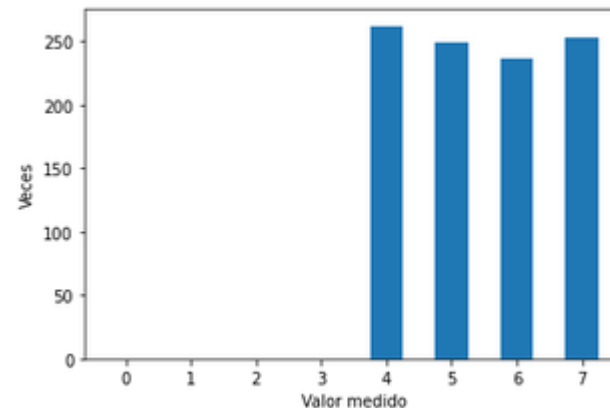
```
[[-3.2828515  2.8567927]]  
El error final es 3.958520267133281e-08
```

$$V_0 = -3.28 \sim 3$$

$$V_1 = 2.85 \sim 3$$

Valores muy similares a los del ruido parametrizado

- Simulación con los parámetros resultado:



b2 es el qubit de Bob

```
r = simulator.simulate(c, resolver)  
b0 = r.measurements['(0, 0)'][0]  
b1 = r.measurements['(0, 1)'][0]  
b2 = r.measurements['(0, 2)'][0]  
res.append(4*b2 + 2*b1 + b0)
```

Siempre se mide  $b_2 = |1\rangle$

# Conclusiones

- Parece ser útil para corregir ruido parametrizado
- Es complicado estimar valores como  $\pi$  por tener infinitos decimales, o se necesitan más nodos en la red neuronal para hacerlo más precisamente