Quantum Differential Machine Learning:

\# as applied to European Options Pricing

```
contents = [
    ‘black-scholes merton’,
    ‘dml’,
    ‘parameter shift rule’,
    ‘embedding’,
    ‘circuit’,
    ‘learning’,
    ‘analysis’
]
```

Options

An option is a legal contract between two parties to buy (call, C) or sell (put, P) an asset at a fixed price (strike price, K) at some deferred date. If the option can only be exercised *at* some fixed date, T, then it is called a European option.

Black-Scholes Equation

To establish a fair price, V, for an option, we assume a principle of no-arbitrage and that asset prices, S, follow Brownian motion to derive the Black Scholes Equation, which relates V with S via a second order partial differential equation. V also depends on two other quantities, the risk free return rate, r, and the volatility, σ of the asset, which are assumed fixed for a European option.

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

The Greeks

The sensitivities of prices, V, with respect to the parameters of the equation, S, T, r, etc. are known as the Greeks collectively.

$$\frac{\partial V}{\partial S} = \Delta \qquad \frac{\partial^2 V}{\partial S^2} = \Gamma \qquad -\frac{\partial V}{\partial \tau} = \Theta \qquad \frac{\partial V}{\partial \sigma} = \mathcal{V}$$

```python
# ANALYTIC BLACK-SCHOLES VALUES
def BS_EuroCall_V(S, K, t, r, σ):
    d1 = (np.log(S/K) + (r + 0.5 * σ**2) * t) / (σ * np.sqrt(t))
    d2 = (np.log(S/K) + (r - 0.5 * σ**2) * t) / (σ * np.sqrt(t))

    V = S * si.norm.cdf(d1) - K * np.exp(-r*t) * si.norm.cdf(d2)

    return V

def BS_EuroCall_Δ(S, K, t, r, σ):
    d1 = (np.log(S/K) + (r + 0.5 * σ**2) * t) / (σ * np.sqrt(t))

    Δ = si.norm.cdf(d1)

    return Δ
```

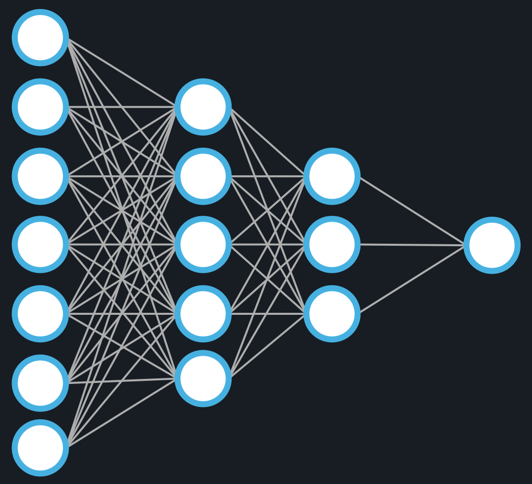
Feedforward

The feedforward mechanism describes the forward flow of information through a neural network.

$$z_0 = x$$

$$z_l = g_{l-1}(z_{l-1}) w_l + b_l$$

$$y = z_L$$



Backpropagation

The backpropagation algorithm describes the backward flow of information used during learning.

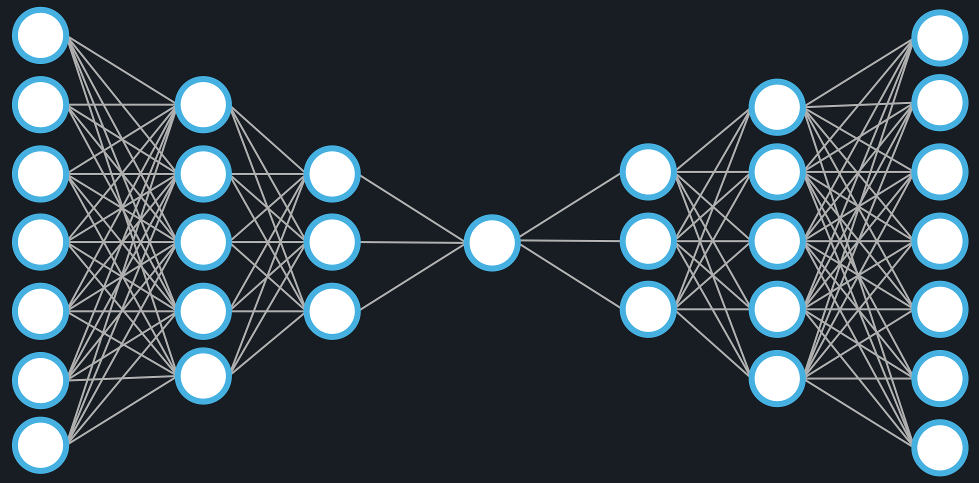
$$\bar{z_L} = \bar{y}$$

$$\overline{z_{l-1}} = \left(\bar{z_l} w_l^T\right) \circ g'_{l-1}(z_{l-1})$$

$$\bar{x} = \bar{z_0}$$

Differential Machine Learning

DML leverages the similarity between backpropagation and feedforward equations by defining a twin network, where backpropagation acts as a second feedforward network. Inputs x are transformed into intermediate y and final output dy/dx, enabling the network to learn the function's behavior, not just its values.



We define the circuit as acting on $|0\rangle$ with a gate $U(\Theta)$, and measure the output via an observable B, we have

$$f(\theta_1, \theta_2, \ldots) = \left\langle 00 \middle| U^\dagger B U \middle| 00 \right\rangle$$

```
# GRADIENTS
def grad(circ, dist, x, param):
    # Market param gradient via paramshift
    # grad = ½(f(x + ½π) - f(x - ½π))
    f_plus = circ(dist, x, param)
    f_minus = circ(dist, x, -param)

    return 0.5 * (f_plus - f_minus)
```

The Parameter Shift Rule

Given a gate $U_P(\Theta) = e^{ia\theta\mathcal{G}}$, for $\theta \in \Theta, a \in \mathbb{R}$, with the matrix $\mathcal{G}$ having exactly two unique eigenvalues e_0 and e_1, we have the following relation ship

$$\frac{\partial f}{\partial \theta} = r\left(f\left(\theta + \frac{\pi}{4r}\right) - f\left(\theta - \frac{\pi}{4r}\right)\right) \text{where } r = \frac{a}{2}(e_1 - e_0)$$

If $\mathcal{G} \in \{X, Y, Z, X \otimes X, Y \otimes Y, Z \otimes Z\}$ then we have $a = \frac{1}{2}$, $e_0 = -1, e_1 = 1,$ and $r = \frac{1}{2}$. Thus, we obtain, for any parameter ϕ of the circuit

$$\frac{\partial f}{\partial \phi} = \frac{f\left(\phi + \frac{\pi}{2}\right) - f\left(\phi - \frac{\pi}{2}\right)}{2}$$

Embedding The Data

The gate parameters, $\Theta = \{\theta_1, \theta_2, \ldots\}$, are implemented via rotation gates, $X, Y, Z, X \otimes X, Y \otimes Y, Z \otimes Z$ which rotate qubits by θ. Similarly, the market parameters are embedded into the circuit by rotation gates which rotate the initial qubit state by $\Phi = \{\phi_1, \phi_2, \ldots\}$. We embed the market parameters S into Φ via the map

$$x = \tanh\left(\frac{X}{a}\right)^b$$

$$\phi = \arcsin(x)$$

We choose tanh because it maps $\mathbb{R}$ to the interval $(-1,1)$, and we use the hyperparameters $(a, b) = (C_1, \beta)$ to embed the spot price, while we use $(a, b) = (C_2, \gamma)$ to embed the expected values.

```
# EMBEDDINGS
def hyp_normalise(x, a, b):
    return np.tanh((x/a)**b)

def hyp_unenmbed(x, a, b):
    return np.power(np.arctanh(x), 1/b) * a
```

The Circuit

The Quantum Neural Network is represented as a function which accepts the parameters Θ which represents the trainable weights of the network along with the input x and the hyperparameter paramshift.

The circuit itself is self-descriptive, using a combination of rotation and entangling gates to process the input.

The IsingXX gate and Strongly Entangling Layers template provide the building blocks to create a highly expressive quantum circuit capable of capturing quantum correlations.

```
# CIRCUIT
n_layers = 3
n_qubits = 2


def qnn(θ, x, paramshift = 0): # Quantum Neural Network

    dev = qml.device("default.qubit", wires=n_qubits)
    # Seperate device for each run since multithreading/multiprocessing intereres

    @qml.qnode(dev, diff_method="parameter-shift")
    def circuit():
        φ = [np.arcsin(x) for i in range(5)]

        φ[np.abs(paramshift) - 1] += np.sign(paramshift) * 0.5 * np.pi

        qml.RY(φ[0],       wires=0)
        qml.RY(φ[1],       wires=1)
        qml.IsingXX(φ[2], wires=[0,1])
        qml.RY(φ[3],       wires=0)
        qml.RY(φ[4],       wires=1)

        qml.templates.StronglyEntanglingLayers(θ, wires=range(n_qubits))

        return qml.expval(qml.PauliZ(0))

    return circuit()
```

Learning

We use an ADAM Optimizer with a learning rate 0.01. We randomly initialise our Θ and keep track of the historic evolution of Θ, and the cost function as applied to both the learning and testing datasets for analytics purposes. We also implement multithreading allowing us to track the behaviour of the circuit with different initial values of Θ. We use MSE between Y and Y_pred to define the cost function.

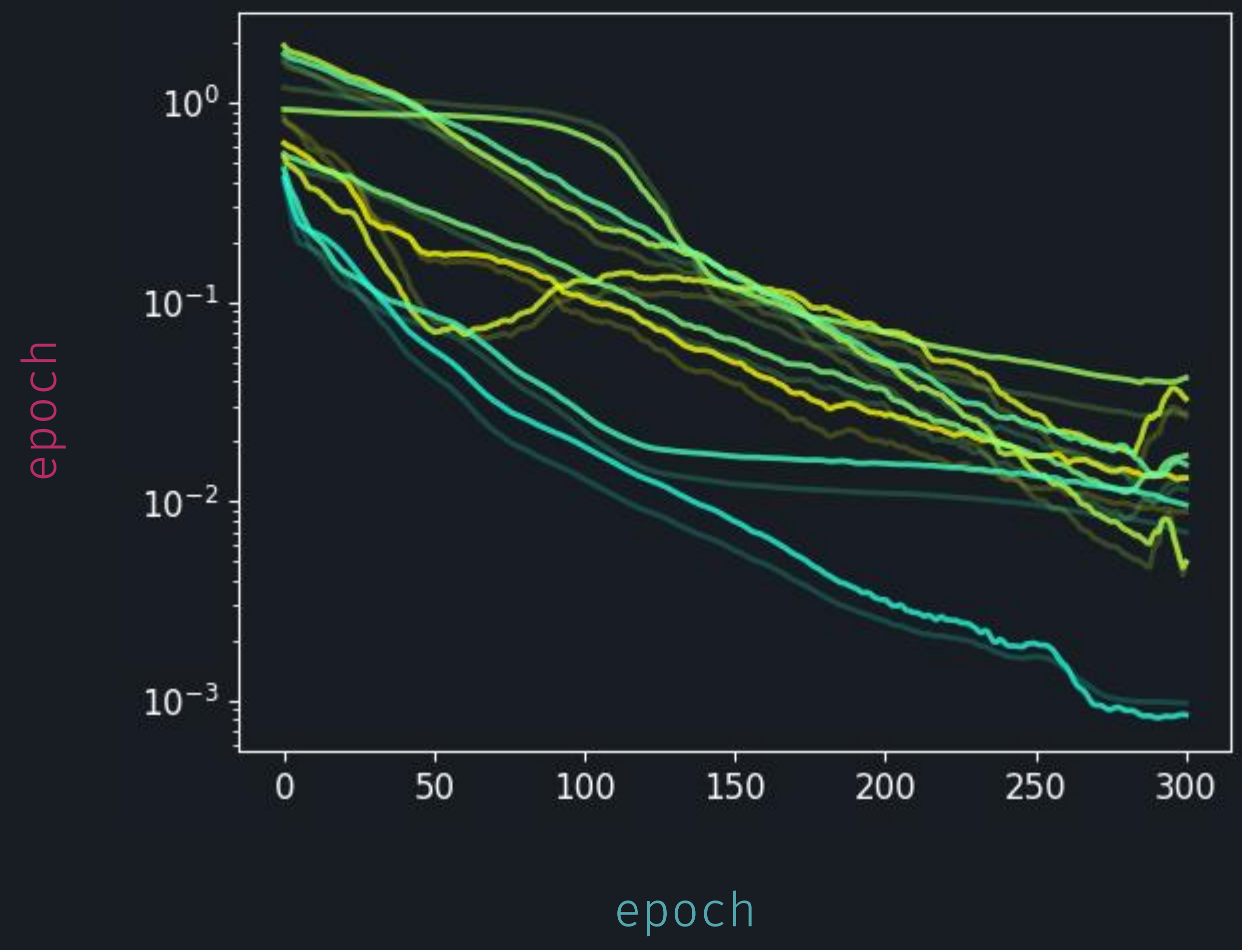
Cost Function

We define the cost function as $C = MSE + \lambda\overline{MSE}$ to allow the QNN to learn both Values and their Deltas.

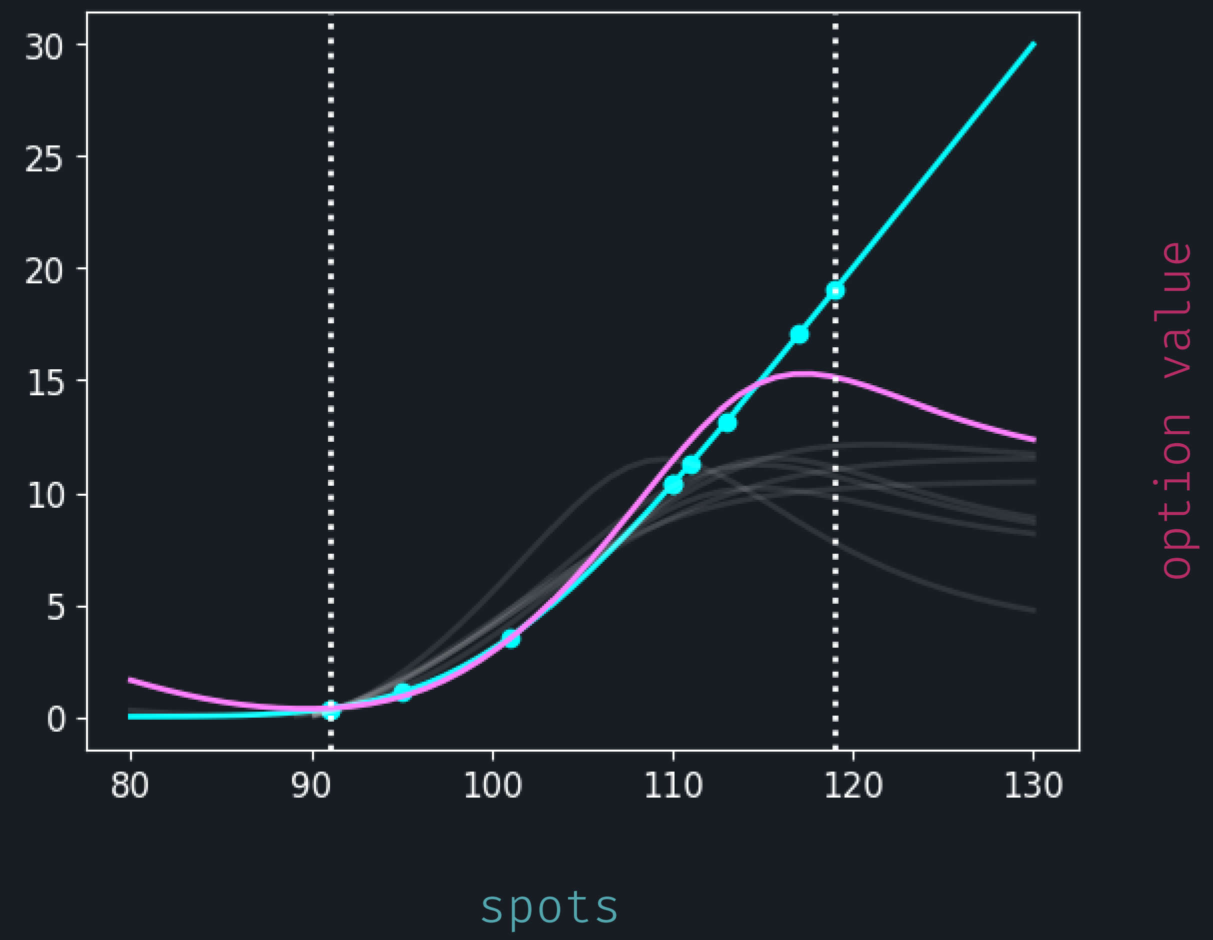
```python
def train(seed, n_L, n_Q, X_training, Y_training, X_testing, Y_testing):

    np.random.seed(seed)
    θ = [np.random.uniform(high = 2*np.pi, size=(n_L, n_Q, 3))]

    historic_training_cost = []
    historic_testing_cost = []

    for epoch in range(301):
        θ_new, _cost = opt.step_and_cost(lambda v: cost(v, X_training, Y_training), θ[epoch])
        _test_cost = cost(θ[epoch], X_testing, Y_testing)

        θ.append(θ_new)

        historic_training_cost.append(_cost)
        historic_testing_cost.append(_test_cost)

    return {
        'seed': seed,
        'θ': θ,                                                                 # full history of θ
        'θ': θ[np.argmin(historic_testing_cost)],                               # optimal θ
        'historic_training_cost': historic_training_cost,                       # full history of training costs
        'historic_testing_cost': historic_testing_cost,                         # full history of testing costs
        'best_index': np.argmin(historic_testing_cost),                         # index of θ in θ, dumb ik
        'best_cost': historic_testing_cost[np.argmin(historic_testing_cost)]    # best cost
    }
```

```python
# COST FUNC

def square_loss(desired, predictions):
    sqr_loss = 0
    for loss, pred in zip(desired, predictions):
        sqr_loss += (loss - pred)**2 / loss
    sqr_loss = sqr_loss / len(desired)
    return sqr_loss

def cost(dist, inputs, labels):
    preds = [qnn(dist, x) for x in inputs]
    return square_loss(labels, preds)
```

```python
def parallel_training(N, n_L, n_Q, X_train, Y_train, X_test, Y_test):

    results = []

    with ThreadPoolExecutor(max_workers=4) as executor:
        futures = [
            executor.submit(
                train,
                seed,
                n_L,
                n_Q,
                X_train,
                Y_train,
                X_test,
                Y_test,
            )

            for seed in range(N)

        ]

        for future in as_completed(futures):
            results.append(future.result())

    return results
```



10^{0}
10^{-1}
10^{-2}
10^{-3}
0
50
100
150
200
250
300
epoch
epoch



30
25
20
15
10
5
0
80
90
100
110
120
130
spots
option value

References

Quantum Differential Machine Learning
T Sakuma, 2023

Differential Machine Learning
B.N. Huge & A. Savine, 2020

Quantum Circuit Learning
K Mitarai *et al.*, 2019

Evaluating Analytic Gradients on Quantum Hardware
M Schuld *et al.*, 2018

Optimal Quantum Circuits for General Two-Qubit Gates
F. Vatan & C. Williams, 2004