Quantum Computing

Chapter 08: Quantum Artificial Intelligence

Prachya Boonkwan

Version: January 14, 2026

Sirindhorn International Institute of Technology
Thammasat University, Thailand

Who? Me?

- Nickname: Arm (P'/N' Arm, etc.)
- Born: Aug 1981
- Work
  - Researcher at NECTEC 2005-2024
  - Lecturer at SIIT, Thammasat University 2025-now
- Education
  - B.Eng & M.Eng in Computer Engineering, Kasetsart University, Thailand
  - Obtained Ministry of Science and Technology Scholarship of Thailand in early 2008
  - Did a PhD in Informatics (AI & Computational Linguistics) at University of Edinburgh, UK from 2008 to 2013

Table of Contents

Variational Quantum Algorithms

Machine Learning

- Machine learning is a class of algorithms and statistical models that make a prediction on an unseen input from the patterns observed in the training dataset
- Supervised learning is a machine learning technique that classifies an unseen item with respect to the criteria, a.k.a. model, constructed from the observed patterns
- The model consists of adjustable parameters, so that it performs accurate classification

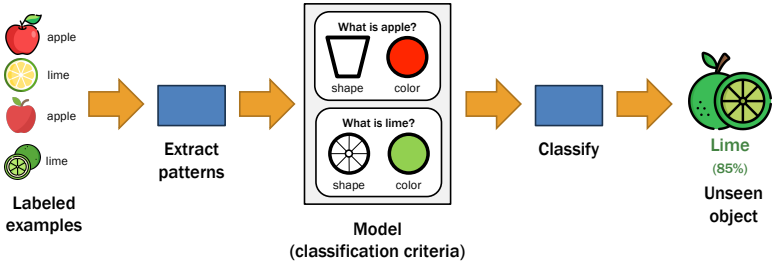


Figure 1: Supervised learning. The model consists of two parameters: shape and color.

Variational Quantum Algorithm

- A class of hybrid quantum algorithms that find approximate solutions to problems with both classical and quantum computing
  - We convert classical data items to equivalent quantum states
  - Parameterized quantum circuit computes a temporary solution
  - We measure the distance between this temporary solution and the gold standard
  - We optimize the parameters to minimize the distance
  - After some iterations, we obtain the optimal solution (parameters) to the problem
- Spoiler: Variational quantum algorithms can be used as activation functions!

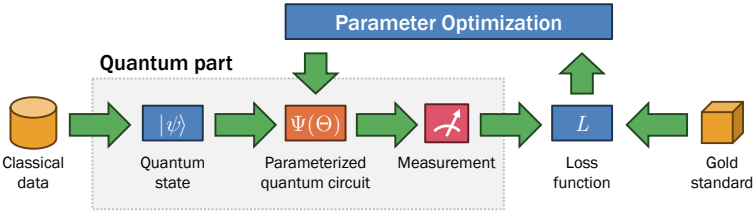


Figure 2: Variational quantum algorithm. Variation refers to a change in condition within certain limits.

Quantum Embedding

- The first step of variational quantum algorithms is to convert a data item into an equivalent quantum state
- Three approaches: direct method, angular encoding, and amplitude encoding
- Basis embedding: We can convert a set of bitstrings into a mixed state by converting each bitstring to an equivalent basis state:

$$\{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \quad \mapsto \quad \frac{1}{\sqrt{M}} \sum_{k=1}^{M} |x_k\rangle$$

We use this method when the input is a set of binary features

- Angular embedding: We can convert a list of real-valued features into its quantum state by phase rotation

$$\{x_1, \ldots, x_N\} \quad \mapsto \quad \left(\prod_{k=1}^{N} \mathrm{RY}(x_k) \right) |0\rangle^{\otimes N}$$

- Amplitude embedding: We can a time series of real-valued items into its quantum state by superposition

$$[x_1, \ldots, x_N] \quad = \quad \sum_{k=0}^{N-1} x_k \, |\mathbb{B}_L(k)\rangle$$

where $L = \lceil \log_2 N \rceil$ is the length of qubits

Quantum Embedding in PennyLane

```
basis_wires = ['d1', 'd2', 'd3']
angular_wires = ['g1', 'g2', 'g3']
amplitude_wires = ['a1', 'a2', 'a3']
dev = qml.device('default.qubit', wires=basis_wires + angular_wires + amplitude_wires)

@qml.qnode(dev, shots=100)
def embedding_circuit():
    # Basis encoding: N binary digits -> N qubits
    bitstring = [1, 0, 1]
    qml.BasisEmbedding(features=bitstring, wires=basis_wires)
    # Angular encoding: N continuous values -> N qubits
    angles = [np.pi / 4, np.pi / 3, np.pi / 2]
    qml.AngleEmbedding(features=angles, wires=angular_wires)
    # Amplitude encoding: 2^N continuous values -> N qubits
    amplitudes = [2.0, 1.0, 3.0, 7.0, 5.0, 4.0, 8.0, 6.0]
    amplitudes = amplitudes / np.linalg.norm(amplitudes)        # Normalization is required
    qml.AmplitudeEmbedding(features=amplitudes, wires=amplitude_wires)
    # Measurements
    return ( qml.counts(wires=basis_wires),
             qml.counts(wires=angular_wires),
             qml.counts(wires=amplitude_wires) )

basis_counts, angular_counts, amplitude_counts = embedding_circuit()
```

Ansatz (n.) ‘guess’ [German]

- In most cases, we design parameterized quantum circuits in accordance with the problem; i.e. we make a guess or problem-inspired ansatz (quantum circuit)
- There are also problem-agnostic ansatzes, e.g. entanglers, but they consist of more quantum gates and parameters
- Example: Consider the following ansatz.

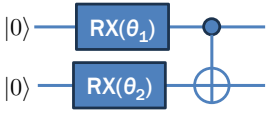


Figure 3: Simple ansatz

$$
\begin{aligned}
&\mathrm{CNOT}^{(2)}_{1,2}\mathrm{RX}^{(2)}_{2}(\theta_2)\mathrm{RX}^{(2)}_{1}(\theta_1)\,|00\rangle \\
&= \mathrm{CNOT}^{(2)}_{1,2}\left(\cos\frac{\theta_1}{2}\,|0\rangle - i\sin\frac{\theta_1}{2}\,|1\rangle\right) \\
&\qquad \otimes \left(\cos\frac{\theta_2}{2}\,|0\rangle - i\sin\frac{\theta_2}{2}\,|1\rangle\right) \\
&= \mathrm{CNOT}^{(2)}_{1,2}\left(\cos\frac{\theta_1}{2}\cos\frac{\theta_2}{2}\,|00\rangle - i\cos\frac{\theta_1}{2}\sin\frac{\theta_2}{2}\,|01\rangle\right. \\
&\qquad \left. - i\sin\frac{\theta_1}{2}\cos\frac{\theta_2}{2}\,|10\rangle - \sin\frac{\theta_1}{2}\sin\frac{\theta_2}{2}\,|11\rangle\right) \\
&= \cos\frac{\theta_1}{2}\cos\frac{\theta_2}{2}\,|00\rangle - i\cos\frac{\theta_1}{2}\sin\frac{\theta_2}{2}\,|01\rangle \\
&\qquad - i\sin\frac{\theta_1}{2}\cos\frac{\theta_2}{2}\,|11\rangle - \sin\frac{\theta_1}{2}\sin\frac{\theta_2}{2}\,|10\rangle
\end{aligned}
$$

Problem-Agnostic Ansatzes/1

- Basic entangler: We rotate each k-th qubit with $\mathbf{RX}(\theta_k)$ and create an entangled state out of them

$$\mathbf{BEL}(\Theta) = \prod_{j=1}^{N} \mathbf{CNOT}^{(N)}_{j,(j+1)\%N} \prod_{k=1}^{N} \mathbf{RX}^{(N)}_{k}(\theta_k)$$

where parameters $\Theta = (\theta_1, \ldots, \theta_N)$

- Recall that $\mathbf{RX}(\theta)$ rotate the qubit about the X-axis

$$|0\rangle \mapsto \cos\frac{\theta}{2}|0\rangle - i\sin\frac{\theta}{2}|1\rangle$$
$$|1\rangle \mapsto -i\sin\frac{\theta}{2}|0\rangle + \cos\frac{\theta}{2}|1\rangle$$

- In one basic-entangler layer for N wires, there are only N parameters
- We can stack L basic-entangler layers up in a more complex model

$$\mathbf{MultiBEL}(\Theta_{L\times N}) = \prod_{k=1}^{L} \mathbf{BEL}(\Theta_k)$$

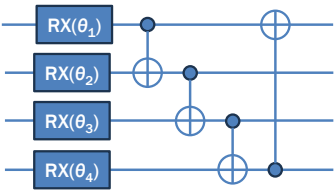


Figure 4: Basic-entangler layer

Problem-Agnostic Ansatzes/2

- Strong entangler layer: We rotate each k-th qubit with $\mathbf{R}(\alpha_k, \beta_k, \gamma_k, \phi)$ and create an entangled state out of them

$$\mathbf{SEL}(\Theta) = \prod_{j=1}^{N} \mathbf{CNOT}^{(N)}_{(j+1)\%N,j} \prod_{k=1}^{N} \mathbf{R}(\alpha_k, \beta_k, \gamma_k, \phi)$$

where parameters $\Theta = (\alpha_1, \beta_1, \gamma_1, \ldots, \phi)$

- $\mathbf{R}(\alpha, \beta, \gamma, \phi)$ rotates the qubit with:

$$|0\rangle \mapsto \operatorname{cis}\beta \cos\alpha\, |0\rangle - \operatorname{cis}(-\gamma) \sin\alpha\, |1\rangle$$
$$|1\rangle \mapsto \operatorname{cis}\gamma \sin\alpha\, |0\rangle + \operatorname{cis}(-\beta) \cos\alpha\, |1\rangle$$

where the global phase term is $\operatorname{cis}\phi$

- In one strong-entangler layer for N wires, there are $3N+1$ parameters
- We can also stack L strong-entangler layers up in a more complex model

$$\mathbf{MultiSEL}(\Theta_{L\times 3N}) = \prod_{k=1}^{L} \mathbf{SEL}(\Theta_k)$$

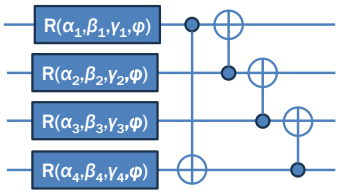


Figure 5: Strong-entangler layer

Expressivity and Trainability

- Expressivity refers to the number of entanglement types an ansatz can generate from its gates and parameters
- Trainability refers to the number of iterations in estimating the optimal parameters from the dataset
- General trend:
  - The more parameters the ansatz has, the more types of entanglement it can generate; i.e. it is more flexible
  - The more parameters the ansatz has, the more iterations it takes to estimate the optimal parameters due to additional randomness; i.e. it is less trainable
- Hardware efficiency refers to ready-made hardware implementation of an ansatz so that it needs not be emulated
- Modularity refers to dividing the entanglement generation into manageable parts (with less parameters) so that the ansatz is more trainable

Entanglers	Basic	Strong
Parameters	$L \times N$	$L \times (3N + 1)$
Efficiency	Maximum	High
Expressivity	Low	Maximum
Trainability	High	Low

Table 1: Comparison of entangler layers

Ansatz Architectures

The design of ansatz architecture can be generally classified into three categories

Hardware-Efficient Ansatz

- Offered by many NISQ device implementations
- High expressivity: more layers = more coverage of quantum states
- Low trainability due to numerous parameters

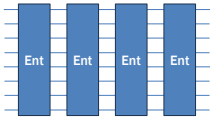


Figure 6: HEA architecture

TENsor product ansatz

- Quantum states are divided into modules for ease of computation
- Low expressivity: less entanglement types
- High trainability due to modularity

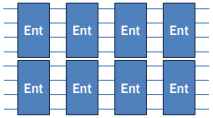


Figure 7: TEN architecture

ALTernating layered ansatz

(a.k.a. brick-wall ansatz)

- Crossing modules overlap one another
- High expressivity: good at entanglements
- Medium trainability due to modularity

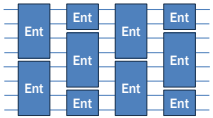


Figure 8: ALT architecture

Ansatz Architectures in PennyLane

```
import pennylane as qml
from pennylane import numpy as np

basic_wires = ['b1', 'b2', 'b3', 'b4']
strong_wires = ['s1', 's2', 's3', 's4']
dev = qml.device('default.qubit', wires=basic_wires + strong_wires)

basic_params = np.random.uniform(0, 2 * np.pi, (2, 4))        # 2 layers, 4 wires
strong_params = np.random.uniform(0, 2 * np.pi, (2, 4, 3))    # 2 layers, 4 wires, 3 angles

@qml.qnode(dev, shots=100)
def circuit(basic_params, strong_params):
    # Basic entangler requires a L x W matrix of weights
    qml.BasicEntanglerLayers(basic_params, wires=basic_wires)
    # Strong entangler requires a L x W x 3 matrix of weights
    qml.StronglyEntanglingLayers(strong_params, wires=strong_wires)
    return ( qml.counts(wires=basic_wires),
             qml.counts(wires=strong_wires) )

basic_counts, strong_counts = circuit(basic_params, strong_params)
print(basic_counts)
print(strong_counts)
```

Exercise 8.1: Variational Quantum Algorithms

Answer the following questions.

1. Explain how we may apply variational quantum algorithms to machine learning. [*Hint*: problem-agnostic ansatzes]
2. Discuss the differences between each quantum embedding method. Which method should be used in which situation? Are there any situations where more than one method can be applied?
3. Discuss the differences of basic and strong entangler layers in terms of parameters, efficiency, expressivity, and trainability.
4. Explain why the alternating layered ansatz seems to be outperform the other architectures.
5. Suppose we have a basic entangler layer of 3 qubits. Compute $\mathbf{BEL}(\mathbf{0})\ |0\rangle^{\otimes 3}$.
6. Suppose we have a basic entangler layer of 3 qubits. Compute $\mathbf{BEL}(\frac{\pi}{2}\mathbf{1})\ |0\rangle^{\otimes 3}$.
7. Suppose we have a strong entangler layer of 3 qubits. Compute $\mathbf{SEL}(\mathbf{0})\ |0\rangle^{\otimes 3}$.
8. Suppose we have a strong entangler layer of 3 qubits. Compute $\mathbf{SEL}(\frac{\pi}{2}\mathbf{1})\ |0\rangle^{\otimes 3}$.

Hybrid Classical-Quantum Machine Learning

Prediction Loss

- Once an ansatz $\mathbf{U}(\Theta)$ is established, we can translate an input data into an output quantum state w.r.t. current parameters
- Challenge 1: We estimate the ansatz's parameters to minimize the prediction loss (the difference between an output quantum state and a gold standard in classical computing)
- Challenge 2: Conversion from classical data to quantum state is usually noisy on NISQ hardware
- Challenge 3: Measurement of a quantum state depends on the observable (i.e. set of orthonormal projection axes)

- Prediction loss: For observables $\{\mathbf{B}_1, \dots, \mathbf{B}_N\}$ and gold standard labels $\{t_1, \dots, t_N\}$, the loss function is

$$L(\Theta) = \sum_{k=1}^{N} f(\mathbf{B}_k, \mathbf{U}(\Theta), t_k)$$

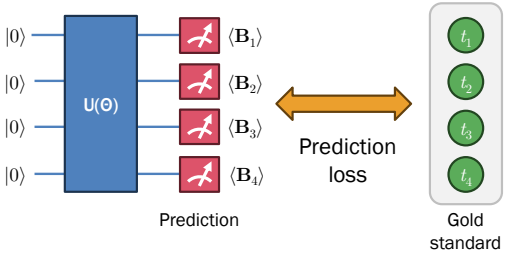


Figure 9: Quantum prediction

Parameter Optimization

- We can approximate the parameters Θ such that the prediction loss L is minimized at the optimal parameters Θ^*
- Local optimum is guaranteed

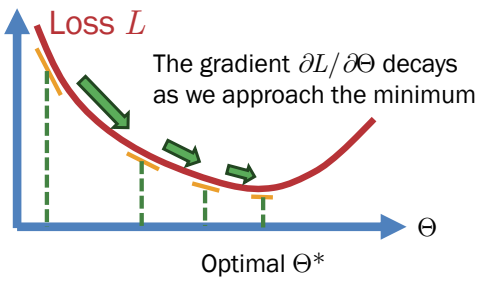


Figure 10: Stochastic gradient descent algorithm

- Stochastic gradient descent (SGD)
  1. Initialize the parameters $\Theta^{(0)}$ with randomization

$$\Theta^{(0)} \sim P_{\text{uniform}}(0, 2\pi)$$

  2. Update the parameters w.r.t. the gradient (slope) at the current point

$$\Theta^{(k+1)} = \Theta^{(k)} - \eta \left. \frac{\partial L}{\partial \Theta} \right|_{\Theta=\Theta^{(k)}}$$

     where the learning rate η is a small constant, e.g. 10^{-4}; this is called the update equation
  3. Repeat step 2 until the parameters converge (i.e. the gradient is close to zero)

Projection-based Loss Functions

- Recall CH4: Suppose that

$$|\psi\rangle \quad = \quad a\,|0\rangle + b\,|1\rangle$$

Projections on popular observables are

$$\begin{aligned}
\left\langle \mathbf{X}^{(\psi)} \right\rangle &= |a+b|^2 - |a-b|^2 \\
\left\langle \mathbf{Y}^{(\psi)} \right\rangle &= |a+bi|^2 - |a-bi|^2 \\
\left\langle \mathbf{Z}^{(\psi)} \right\rangle &= |a|^2 - |b|^2 \\
\left\langle \mathbf{I}^{(\psi)} \right\rangle &= |a|^2 + |b|^2 \quad = \quad 1
\end{aligned}$$

- Projection on $\mathbf{Z}$ is the most popular because it points towards $|0\rangle$ (positive value) and $|1\rangle$ (negative value)

- Our ansatz makes prediction by taking $|0\rangle^{\otimes N}$ as an input

$$|\psi\rangle \quad = \quad \mathbf{U}(\Theta)\,|0\rangle^{\otimes N}$$

- Letting $\mathbf{B} \in \{\mathbf{X}, \mathbf{Y}, \mathbf{Z}\}$ be a basis, projection on $\mathbf{B}$ is

$$\begin{aligned}
\left\langle \mathbf{B}^{(\psi)} \right\rangle &= \langle\psi|\ \mathbf{B}\ |\psi\rangle \\
&= \left[\langle 0|^{\otimes N}\,\mathbf{U}^*(\Theta)\right] \mathbf{B} \left[\mathbf{U}(\Theta)\,|0\rangle^{\otimes N}\right]
\end{aligned}$$

or in abbreviation

$$\left\langle \mathbf{B}^{(\psi)} \right\rangle \quad = \quad \langle 0|\,\mathbf{U}^*(\Theta)\ \mathbf{B}\ \mathbf{U}(\Theta)\,|0\rangle$$

Projection-based Loss Functions (cont'd)

- We define our loss function as projection on the basis $\mathbf{B}_k$ as follows

$$\begin{aligned} & f(\mathbf{B}_k, \mathbf{U}(\Theta), t_k) \\ = \; & \Delta\left(\left\langle \mathbf{B}_k^{(\mathbf{U}(\Theta)|0\rangle)} \right\rangle, t_k\right) \\ = \; & \Delta\left(\langle 0|\,\mathbf{U}^*(\Theta)\,\mathbf{B}_k\,\mathbf{U}(\Theta)\,|0\rangle, t_k\right) \end{aligned}$$

  where $\Delta(b_k, t_k)$ is a distance function between the expected measurement b_k and the gold standard label t_k
- Since the expectation $b_k \in [-1, 1]$, each gold-standard label is $t_k \in \{-1, 1\}$

- Criteria for prediction loss
  - Faithfulness: The minimum of prediction loss should correspond to the solution of the problem
  - Efficiency: We should be able to efficiently estimate the prediction loss by quantum measurement, perhaps accompanied with classical postprocessing
  - Meaningfulness: Smaller prediction loss should indicate better quality of the solution
  - Trainability: The prediction loss should be possible to efficiently optimize the parameters ⇒ preferably differentiable by the parameters Θ

Popular Loss Functions/1

- Mean squared error (MSE) measures the squared differences between each expected measurement b_k and the corresponding gold standard label t_k

$$\Delta_{\text{MSE}} = \frac{1}{N}\sum_{k=1}^{N}\left(t_k - b_k\right)^2$$

- The gradient (slope) of MSE is derived by

$$\begin{aligned}\frac{\partial}{\partial\Theta}\Delta_{\text{MSE}} &= \frac{\partial}{\partial\Theta}\frac{1}{N}\sum_{k=1}^{N}\left(t_k - b_k\right)^2 \\ &= -2\sum_{k=1}^{N}\left(t_k - b_k\right)\frac{\partial b_k}{\partial\Theta}\end{aligned}$$

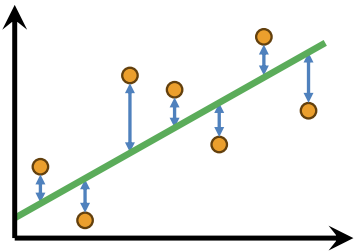


Figure 11: Mean squared error measures the differences between prediction (green line) and the gold standard label (yellow dots). These distances are counted as loss.

Popular Loss Functions/2

- Negative Log-Likelihood (NLL) measures the likelihood of the gold-standard labels according to the current model

$$\Delta_{\text{NLL}} = -\frac{1}{N}\sum_{k=1}^{N}\log\left(\varpi_M(b_k)\right)$$

- The gradient of NLL is derived by

$$\begin{aligned}\frac{\partial}{\partial\Theta}\Delta_{\text{NLL}} &= -\frac{\partial}{\partial\Theta}\frac{1}{N}\sum_{k=1}^{N}\log\left(\varpi_M(b_k)\right)\\ &= \frac{1}{N}\sum_{k=1}^{N}(\varpi_M(b_k)-t_k)\frac{\partial b_k}{\partial\Theta}\end{aligned}$$

- Softmax distribution for M-dim vector

$$\varpi_M(b_k) = \frac{\exp(b_k)}{\sum_{j=1}^{M}\exp(b_j)}$$

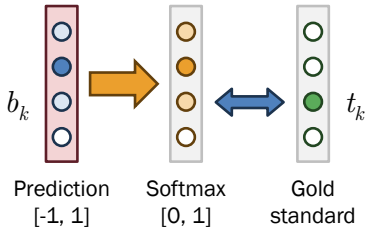


Figure 12: Negative log-likelihood measures the likelihood of the gold-standard labels (green dot) according to the model (yellow dot).

Popular Loss Functions/3

- Hinge loss (HL) measures the sign differences between each expected measurement b_k and the corresponding gold standard label t_k

$$\Delta_{\text{HL}} = \frac{1}{N}\sum_{k=1}^{N}\begin{cases}0 & \operatorname{sgn} t_k = \operatorname{sgn} b_k \\ 1 - t_k b_k & \text{otherwise}\end{cases}$$

- The gradient of HL is derived by

$$\frac{\partial}{\partial\Theta}\Delta_{\text{HL}} = \frac{\partial}{\partial\Theta}\frac{1}{N}\sum_{k=1}^{N_{\text{diff-signs}}}\left(1 - t_k b_k\right)$$

$$= -\frac{1}{N}\sum_{k=1}^{N_{\text{diff-signs}}} t_k \frac{\partial b_k}{\partial\Theta}$$

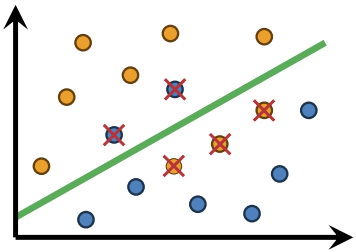


Figure 13: Hinge loss measures the errors of class boundary (green line) and the gold standard labels (yellow and blue dots). The crossed dots denote sign misprediction and are counted as loss.

Hybrid Classical-Quantum Model

- Hybrid model is a variational algorithm, whose parameters are estimated classically
  - Linear regression: learn a linear trend of the dataset; i.e. linear relationship between the input features and the output scalar values
  - Logistic regression: learn to put lables from the input features using the logistic function
  - Support vector machine: learn to draw a class boundary with a non-linear function (kernel)

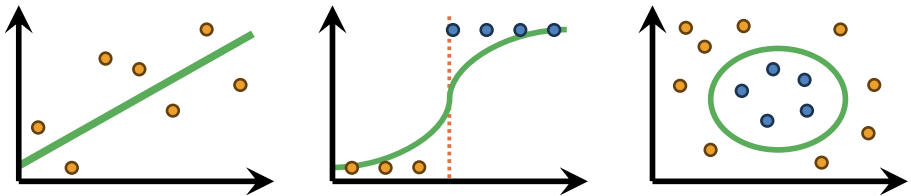


Figure 14: Classical machine learning

Parameter Shift Rule (PSR)

- Challenge: How can we compute

$$\frac{\partial b_k}{\partial \Theta} = \frac{\partial}{\partial \Theta} \langle 0| \mathbf{U}^*(\Theta) \mathbf{B}_k \mathbf{U}(\Theta) |0\rangle$$

- First, let's assume the ansatz $\mathbf{U}$ can be rewritten as

$$\mathbf{U}(\Theta) = \mathrm{cis}\left(-\frac{\Theta}{2}\mathbf{P}\right)$$

where $\mathbf{P}$ is the Hermitian generator of $\mathbf{U}$

- The gradients of $\mathbf{U}$ and $\mathbf{U}^*$ are:

$$\frac{\partial}{\partial \Theta}\mathbf{U}(\Theta) = -\frac{i}{2}\mathbf{P}\mathbf{U}(\Theta) = -\frac{i}{2}\mathbf{U}(\Theta)\mathbf{P}$$
$$\frac{\partial}{\partial \Theta}\mathbf{U}^*(\Theta) = \frac{i}{2}\mathbf{P}\mathbf{U}^*(\Theta) = \frac{i}{2}\mathbf{U}^*(\Theta)\mathbf{P}$$

- Now let's compute the gradient

$$\begin{aligned}
\frac{\partial b_k}{\partial \Theta} &= \frac{\partial}{\partial \Theta} \langle 0| \mathbf{U}^*(\Theta) \mathbf{B}_k \mathbf{U}(\Theta) |0\rangle \\
&= \langle 0| \left(\frac{\partial}{\partial \Theta} \mathbf{U}^*(\Theta) \mathbf{B}_k \mathbf{U}(\Theta) \right) |0\rangle \\
&= \langle 0| \frac{i}{2} \left(\mathbf{U}^*(\Theta)\mathbf{P}\mathbf{B}_k\mathbf{U}(\Theta) \right. \\
&\qquad \left. - \mathbf{U}^*(\Theta)\mathbf{B}_k\mathbf{P}\mathbf{U}(\Theta) \right) |0\rangle \\
&= \langle 0| \left(\frac{i}{2} \mathbf{U}^*(\Theta)[\mathbf{P}, \mathbf{B}_k]\mathbf{U}(\Theta) \right) |0\rangle
\end{aligned}$$

- Since $\mathbf{B}_k$ is a Pauli operator, the identity is:

$$\begin{aligned}
[\mathbf{P}, \mathbf{B}_k] = -i \left(\mathbf{U}^*(\tfrac{\pi}{2}) \mathbf{B}_k \mathbf{U}(\tfrac{\pi}{2}) \right. \\
\left. - \mathbf{U}^*(-\tfrac{\pi}{2}) \mathbf{B}_k \mathbf{U}(-\tfrac{\pi}{2}) \right)
\end{aligned}$$

Parameter Shift Rule (cont'd)

- We then substitute such identity to the gradient and obtain

$$\begin{aligned}\frac{\partial b_k}{\partial \Theta} &= \frac{1}{2}\langle 0|\left(\mathbf{U}^*(\Theta+\frac{\pi}{2})\mathbf{B}_k\mathbf{U}(\Theta+\frac{\pi}{2})\right)|0\rangle \\ &\quad -\frac{1}{2}\langle 0|\left(\mathbf{U}^*(\Theta-\frac{\pi}{2})\mathbf{B}_k\mathbf{U}(\Theta-\frac{\pi}{2})\right)|0\rangle \\ &= \frac{1}{2}\left(b_k(\Theta+\frac{\pi}{2})-b_k(\Theta-\frac{\pi}{2})\right)\end{aligned}$$

  where the expected measurement $b_k(\Theta)$ is

$$b_k(\Theta) = \langle 0|\left(\mathbf{U}^*(\Theta)\mathbf{B}_k\mathbf{U}(\Theta)\right)|0\rangle$$

- Parameter shift rule is irreversible due to projection-based measurement

Gradients of Popular Loss Functions

- Gradient of Mean Squared Error:

$$\frac{\partial}{\partial \Theta} \Delta_{\text{MSE}}(\Theta) = \sum_{k=1}^{N} \left(b_k(\Theta) - t_k\right) \left(b_k(\Theta + \frac{\pi}{2}) - b_k(\Theta - \frac{\pi}{2})\right)$$

- Gradient of Negative Log-Likelihood:

$$\frac{\partial}{\partial \Theta} \Delta_{\text{NLL}}(\Theta) = \sum_{k=1}^{N} \left(\varpi(b_k(\Theta)) - t_k\right) \left(b_k(\Theta + \frac{\pi}{2}) - b_k(\Theta - \frac{\pi}{2})\right)$$

- Gradient of Hinge Loss:

$$\frac{\partial}{\partial \Theta} \Delta_{\text{HL}}(\Theta) = -\frac{1}{2N} \sum_{k=1}^{N_{\text{diff-signs}}} t_k \left(b_k(\Theta + \frac{\pi}{2}) - b_k(\Theta - \frac{\pi}{2})\right)$$

Parameter Optimization in PennyLane

```
import torch as T
from torch import nn as N
from torch import optim as O

wires = ['w1', 'w2', 'w3', 'w4']
dev = qml.device('default.qubit', wires=wires)

@qml.qnode(dev)
def circuit(inputs, params):                          # Quantum circuit
    qml.AngleEmbedding(inputs, wires=wires)           # Encode each input as a quantum state
    qml.BasicEntanglerLayers(params, wires=wires)     # Ansatz
    return [qml.expval(qml.PauliZ(wires=w)) for w in wires]    # Each dimension is in [-1, +1]

class HybridModel(N.Module):                          # Hybrid classical-quantum model
    def __init__(self):
        super().__init__()
        params_shapes = {"params": (3, 4)}    # Define the shape of parameters `params`
        self.circuit_layer = qml.qnn.TorchLayer(circuit, params_shapes)
        self.softmax = N.Softmax(dim=0)
    def forward(self, inputs):
        outvec = self.circuit_layer(inputs)
        outvec = self.softmax(outvec)
        return outvec
```

Parameter Optimization in PennyLane (cont'd)

```
invec = T.tensor([0.0, 0.0, 0.0, 0.0])             # Input vector
goldstd = T.tensor([0.5, 0.0, 0.5, 0.0])           # Gold-standard output vector

hybrid_model = HybridModel()                       # Our ansatz
loss_fn = N.MSELoss()                              # The loss function
opt = O.Adam(hybrid_model.parameters(), lr=0.1)    # SGD algorithm with adaptive gradients
no_iters = 100                                     # Number of iterations

# Make prediction before training
outvec = hybrid_model(invec)
print(f'Before: {outvec}')

# Training loop
for i in range(no_iters):
    outvec = hybrid_model(invec)                   # Make prediction
    loss = loss_fn(outvec, goldstd)                # Compute prediction loss
    opt.zero_grad()                                # Re-estimate the parameters
    loss.backward()
    opt.step()

# Make prediction after training
outvec = hybrid_model(invec)
print(f'After: {outvec}')
```

Exercise 8.2: Hybrid Classical-Quantum Machine Learning

Answer the following questions.

1. What is prediction loss? Explain its relevance to the VQA.
2. Discuss why the stochastic gradient descent algorithm is guaranteed to achieve only a local optimum, not a global one. [*Hint*: Think about a mountain range.]
3. Explain how the parameter shift rule can be used to compute the gradient in the SGD algorithm.
4. Discuss the differences between three kinds of prediction loss: MSE, NLL, and HL. In which situation each prediction loss can be employed?
5. Investigate if the softmax distribution ϖ can be used to convert any arbitrary real-valued vector to a probability distribution. Explain your rationale.
6. Discuss the differences between the negative log-likelihood and the hinge loss. Analyze how they are computed.
7. Review what and how these models learn from the dataset: linear regression, logistic regression, and support vector machine.
8. Explain why the parameter shift rule is irreversible.

Coding Challenge 8.2: Hybrid Classical-Quantum Machine Learning (cont'd)

Implement the following hybrid models using PennyLane. Generate your own randomized datasets for training and testing.

1. Linear regression for 10-dimensional data

$$y = \theta_0 + \sum_{k=1}^{10} \theta_k x_k$$

where each x_k is the k-th item of the data point $\mathbf{x}$, and θ_k is the k-th parameter. Use MSE as the prediction loss.

2. Logistic regression for 10-dimensional data

$$y = \sigma\left(\theta_0 + \sum_{k=1}^{10} \theta_k x_k\right)$$

where each x_k is the k-th item of the data point $\mathbf{x}$, and θ_k is the k-th parameter. Use NLL as the prediction loss.

Quantum Kernel Methods

Class Separation and Kernel Functions

- Class separation refers to how distinct different classes within data, measured by a distance function
- Kernel function $k(\mathbf{x}, \mathbf{x}')$ measures such distance between two data items $\mathbf{x}$ and $\mathbf{x}'$, and its computation can be simplified as a feature map $\Phi : \mathbb{R}^N \mapsto \mathbb{R}^V$, where

$$k(\mathbf{x}, \mathbf{x}') \quad = \quad \langle \Phi(\mathbf{x}), \Phi(\mathbf{x}') \rangle$$

  V is the number of features, and $\langle \mathbf{u}, \mathbf{v} \rangle$ is the inner product of $\mathbf{u}$ and $\mathbf{v}$
- Limitation: Time complexity is $O(N^2 V M)$, where N is the number of input features, and M is the number of training iterations

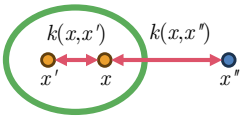


Figure 15: Kernel function

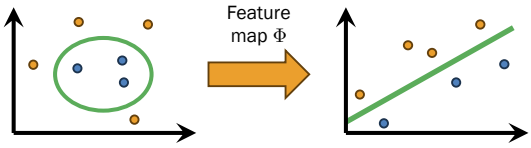


Figure 16: Feature map converts a space of input data with non-linear class bounary to an equivalent feature space with linear class boundary.

Support Vector Machine

- Support vectors are the data points nearest to the class boundary, and the margin is the total distance between these support vectors and the class boundary
- Support vector machines (SVMs) are supervised max-margin models that learn classification and regression, once each data point $\mathbf{x}_k$ is mapped to the feature space with linear class boundary

$$\hat{y}_k = \mathbf{w}^\top \Phi(\mathbf{x}_k) + b$$

where Φ is a feature map function, and $\mathbf{w}$ are model parameters

- The loss function of SVMs is

$$L(\mathbf{w}) = |\mathbf{w}|_2^2 + \alpha \Delta_{\text{HL}}(\mathbf{y}, \hat{\mathbf{y}})$$

where hyperparameter α determines the trade-off between margin size ($0 \leq \alpha < 1$) and more strict prediction ($\alpha > 1$)

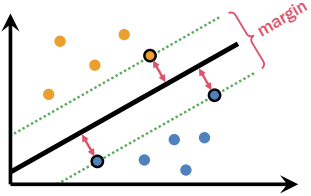


Figure 17: Support vector machine

Quantum Kernel Functions

- We can treat a feature map as a VQA

$$\mathbf{U}_{\Phi}(\mathbf{x}) \quad = \quad \prod_{k=1}^{M} \mathbf{H}^{\otimes V} \Lambda(\mathbf{x})$$

where the unitary operator $\Lambda(\mathbf{x})$ is an angular embedding operator

$$\Lambda(\mathbf{x}) \quad = \quad \bigotimes_{s=1}^{V} \mathbf{RZ}(\phi_s(\mathbf{x}))$$

and each $\phi_s(\mathbf{x})$ is the s-th feature extraction function

- Observe that $\mathbf{U}_{\Phi}(\mathbf{x})$ is untrainable because each $\mathbf{x}_k$ is a constant data point

- We then assemble a quantum SVM by incorporating $\mathbf{U}_{\Phi}$ with an entangler layer:

$$\mathbf{U}_{\text{QSVM}}(\mathbf{x}) \quad = \quad \mathbf{MultiBEL}(\Theta_{L\times V})\, \mathbf{U}_{\Phi}(\mathbf{x})$$

- We compute the measurement operator $\mathbf{M}_y$ for the gold standard by

$$\mathbf{M}_y \quad = \quad |0\rangle^{\otimes V} \langle 0|^{\otimes V}$$

which means we will make measurement only on the first qubit

- Probability of outcome y is therefore

$$p(y|\mathbf{x}) \quad = \quad \left\langle \mathbf{U}^{*}_{\text{QSVM}}(\mathbf{x}) \mid \mathbf{M}_y \mid \mathbf{U}_{\text{QSVM}}(\mathbf{x}) \right\rangle$$

Quantum Support Vector Machine

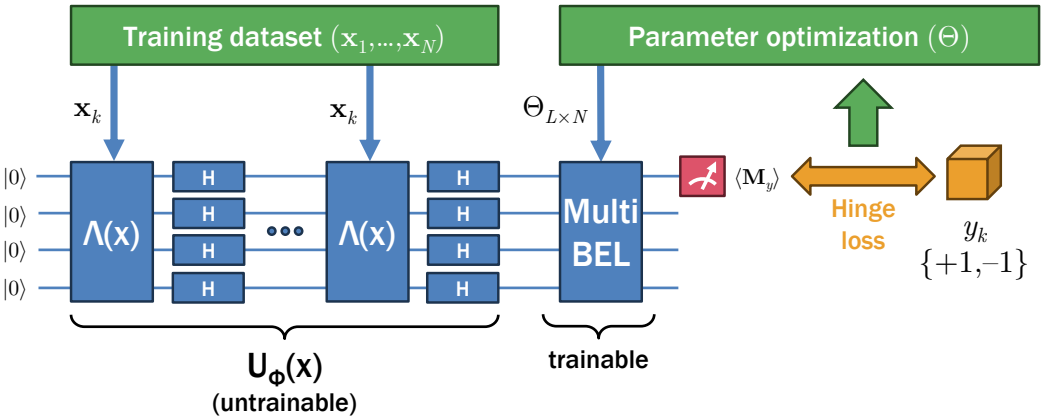


Figure 18: Quantum support vector machine. $\Lambda(\mathbf{x})$ is an angular embedding operator for $\mathbf{x}$. Time complexity for quantum SVM is $O((\log N)^2 VM)$, where N is the number of input features, V is the dimension of feature space, and M is the number of training iterations.

Quantum SVM in PennyLane/1

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

no_qubits = 4
wires = ['x1', 'x2', 'x3', 'x4']
dev = qml.device('lightning.qubit', wires=wires)

########################################
# Dataset Preparation
########################################

def prepare_dataset(no_training, no_testing):
    # Non-linear regression: y = x1 + x2^2 + x3^3 + x4^4 - x0
    # Output: +1 for y >= 0; -1, otherwise
    xs = 2 * np.random.random((no_training + no_testing, 4))
    for i in range(1, no_qubits):
        xs[:, i] = xs[:, i] ** (i + 1)
    coeffs = np.random.random(4)
    threshold = np.random.random() * 6
    ys = np.where(xs @ coeffs - threshold >= 0, +1, -1)
    return xs[:no_training], ys[:no_training], xs[no_training:], ys[no_training:]

xs_train, ys_train, xs_test, ys_test = prepare_dataset(no_training=200, no_testing=50)
```

Quantum SVM in PennyLane/2

```
########################################
# Quantum Support Vector Machine
########################################

def smv_layer(ftrs, params, wires):
    for wire in wires:
        qml.Hadamard(wires=wire)
    qml.AngleEmbedding(ftrs, wires=wires)
    qml.BasicEntanglerLayers(params, wires=wires)

def ansatz(ftrs, params, wires):
    for j in range(params.shape[0]):
        smv_layer(ftrs, params[j], wires)

@qml.qnode(dev)
def kernel_circuit(ftrs1, ftrs2, params):    # Distance between feature vectors ftrs1 and ftrs2
    ansatz(ftrs1, params, wires=wires)
    qml.adjoint(ansatz)(ftrs2, params, wires=wires)
    return qml.probs(wires=wires)

def kernel(ftrs1, ftrs2, params):    # We separate the kernel function from the quantum circuit
    probs = kernel_circuit(ftrs1, ftrs2, params)
    return probs[0]
```

Quantum SVM in PennyLane/3

```
########################################
# Training & Testing
########################################

def kernel_matrix(A, B, params):        # Distance between each pair of training items
    return np.array([[kernel(a, b, params) for b in B] for a in A])

def prepare_params(no_svmlayers, no_entlayers, no_wires):
    dims = (no_svmlayers, no_entlayers, no_wires)
    params = np.random.uniform(0, 2 * np.pi, dims, requires_grad=True)
    return params

no_svmlayers = 3
no_entlayers = 2
init_params = prepare_params(no_svmlayers, no_entlayers, no_qubits)
quantum_kernel = lambda A, B: kernel_matrix(A, B, init_params)

svm = SVC(kernel=quantum_kernel)        # Training
svm.fit(xs_train, ys_train)

predictions = svm.predict(xs_test)      # Prediction
print(f'Accuracy = {accuracy_score(predictions, ys_test)}')
```

Exercise 8.3: Quantum Support Vector Machines

1. Explain how a kernel function measures the distance between two data items using a feature map. What enables non-linearity?
2. Explain what support vectors are and why we want to maximize the margin of class separation in support vector machines.
3. Examine the quantum feature map $\mathbf{U}_\Phi(\mathbf{x})$. Discuss why we encode the features with the angular embedding operator and why we apply the Hadamard gates afterwards.
4. Examine the PennyLane code of QSVM. We have to precompute the kernel matrix, where the distance between each pair of training items is measured with our quantum circuit. What is its time complexity for this step?
5. Discuss the quantum advantage of QSVM. What helps improve the training speed of classical SVM?

Quantum Neural Networks

Classical Perceptron

- Perceptron is a mathematical model that imitates a neuron
- Step 1: Data item q is encoded as a real-valued vector, or feature map

$$\mathbf{f} = E(q)$$

where E is an encoding function

- Step 2: Each feature f_k is multiplied by a connection weight w_k, summed as as a weighted average, and penalized by the input bias λ:

$$u = \sum_{k=1}^{N} w_k f_k - \lambda = \mathbf{w}^\top \mathbf{f} - \lambda$$

- Step 3: We produce an output value

$$y = \alpha(u - \delta)$$

where $\alpha(\cdot)$ is an activation function and δ is the threshold parameter

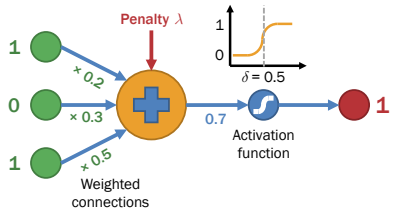


Figure 19: Perceptron

Quantum Perceptron

- In quantum perceptron, we do NOT need any additional activation function!
- Step 1: Each data item q_k is encoded

$$|x_1 \dots x_N\rangle = \bigotimes_{k=1}^{N} \mathbf{RY}(q_k)$$

- Step 2: Each feature $|x_k\rangle$ is rotated about the X axis by phase θ_k:

$$|y\rangle = \bigotimes_{k=1}^{N} \mathbf{RX}(\theta_k)|x_k\rangle$$

where each θ_k is a parameter

- Note that this is tensor product $\otimes$

- In classical computing, we have to incorporate the activation function, e.g. Sigmoid $\sigma(\cdot)$, hyperbolic tangent $\tanh$, and rectified linear unit ReLU, because the weighted average is only linearly learnable
- However, we can treat each rotation in quantum perceptrons as a periodic activation function

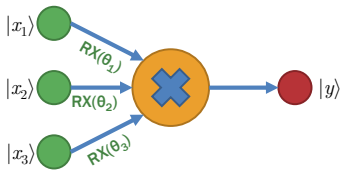


Figure 20: Quantum perceptron

Quantum Multilayer Perceptron (QMLP)

- Each layer consists of multiple quantum perceptrons

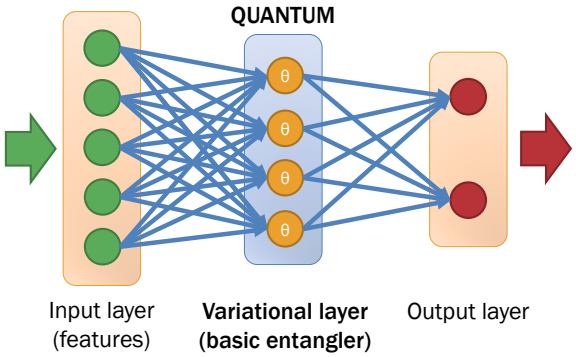


Figure 21: Quantum multilayer perceptron (QMLP)

- Input layer is the data encoder, not an actual group of quantum perceptrons
- Variational hidden layer learns non-linear patterns from the dataset with rotation gates
- Output layer learns to compute the score for each output class from the learned patterns
- Activation functions are NOT necessary because the hidden layer already performs as an activation function

Quantum Deep Neural Networks (QDNNs)

- Learning from multiple levels of representation

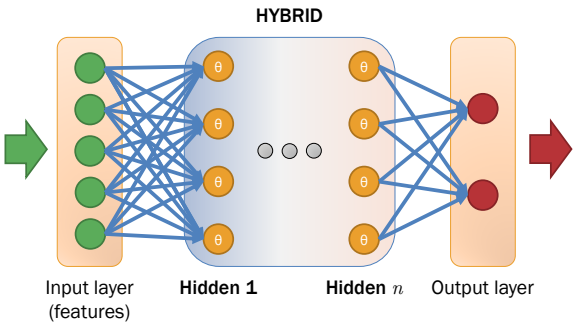


Figure 22: Quantum deep neural networks (QDNNs)

- Each hidden layer can be either classical or variational
  - Classical layer learns linear transformation
  - Variational layer learns an activation function
- The more layers, the more abstract representation it can learn from data
- This results in higher accuracy
- Activation functions are NOT necessary because the hidden layer already performs as one

Training QDNNs with Backpropagation Algorithm

1. Feed a new input data item to the QDNNs and let them produce an output state
2. Measure the prediction loss

$$L = \sum_{k=1}^{N} \Delta(\langle 0| \mathbf{U}^*(\Theta)\, \mathbf{B}_k\, \mathbf{U}(\Theta) |0\rangle, t_k)$$

3. For each layer from the last to the first, we re-estimate each of its parameters θ:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \left. \frac{\partial L}{\partial \theta} \right|_{\theta=\theta^{(k)}}$$

4. Repeat Steps 1-3 until convergence

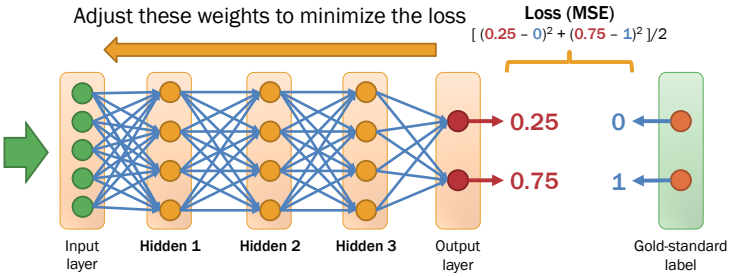
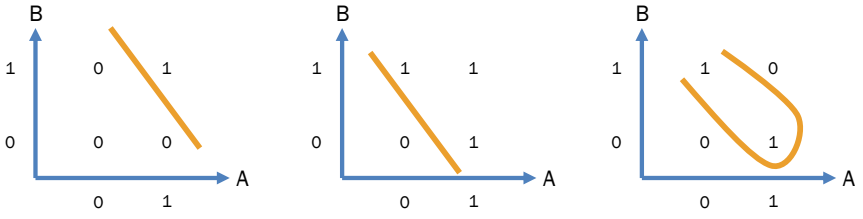


Figure 23: Backpropagation algorithm

Example: XOR Model

- In classical logics, XOR gate poses a challenge in machine learning because its class boundary is non-linear, unlike AND and OR
- We will develop a QMLP model that imitates the XOR gate



AND
Both arguments must be true

OR
At least one argument must be true

XOR
(exclusive OR)
Only one of the arguments must be true

Figure 24: Logic gates and class boundaries

XOR Model in PennyLane/1

```
import torch as T
from torch import nn as N
from torch import optim as O

wires = ['w1', 'w2', 'w3', 'w4']
dev = qml.device('default.qubit', wires=wires)

##### Dataset Preparation #####

def prepare_dataset(no_training, no_testing):
    xor_tbl = T.tensor([ (0, 0, 0),
                         (0, 1, 1),
                         (1, 0, 1),
                         (1, 1, 0) ], dtype=T.float)
    training_idxs = np.random.randint(4, size=no_training)
    testing_idxs = np.random.randint(4, size=no_testing)
    training_data = xor_tbl[training_idxs]
    testing_data = xor_tbl[testing_idxs]
    return training_data, testing_data

no_training = 1000
no_testing = 100
training_data, testing_data = prepare_dataset(no_training, no_testing)
```

XOR Model in PennyLane/2

```
##### Hybrid Multilayer Perceptron #####

@qml.qnode(dev)
def circuit(inputs, params):
    qml.AngleEmbedding(inputs, wires=wires)
    qml.BasicEntanglerLayers(params, wires=wires)
    outvec = [qml.expval(qml.PauliZ(wires=w)) for w in wires]
    return outvec

class HybridMLP(N.Module):
    def __init__(self, dim_input, dim_output):
        super().__init__()
        (self.dim_input, self.dim_output) = (dim_input, dim_output)
        self.linear1 = N.Linear(self.dim_input, 4)
        self.qlayer = qml.qnn.TorchLayer(circuit, {"params": (3, 4)})
        self.linear2 = N.Linear(4, self.dim_output)
    def forward(self, inputs):
        outvec = self.linear1(inputs)           # Layer 1: Linear(2 -> 4)
        outvec = self.qlayer(outvec)            # Layer 2: Basic entangler
        outvec = self.linear2(outvec)           # Layer 3: Linear(4 -> 1)
        return outvec

xor_model = HybridMLP(dim_input=2, dim_output=1)
```

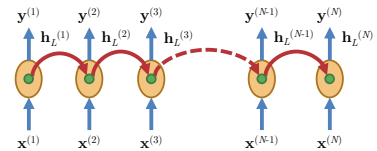
XOR Model in PennyLane/3

```
##### Training #####

loss_fn = N.MSELoss()                              # Mean squared error
opt = O.Adam(xor_model.parameters(), lr=0.1)       # parameters() returns a set of model parameters
no_iters = 30
batch_size = 20
loss_threshold = 1e-5

for i in range(no_iters):                          # Loop of epochs
    np.random.shuffle(training_data)
    total_loss = 0.0
    for j in range(no_training // batch_size):     # Batch training
        batch = training_data[j * batch_size : (j + 1) * batch_size]
        inmat = batch[:, 0:2]
        outmat = xor_model(inmat).flatten()        # Make prediction
        goldmat = batch[:, 2]
        loss = loss_fn(outmat, goldmat)            # Compute prediction loss
        total_loss += loss.detach().data.item()
        opt.zero_grad()                            # Perform backpropagation
        loss.backward()
        opt.step()
    print(f'Total loss {i}: {total_loss}')
    if total_loss < loss_threshold: break
```

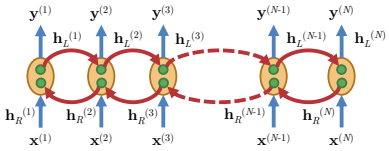
XOR Model in PennyLane/4

```
##### Testing #####

no_correct = 0

for j in range(no_testing // batch_size):
    batch = testing_data[j * batch_size : (j + 1) * batch_size]
    inmat = batch[:, 0:2]
    goldmat = batch[:, 2] >= 0.5
    outmat = xor_model(inmat).flatten()
    preds = outmat >= 0.5
    no_correct += (preds == goldmat).sum().item()

print(f'Accuracy: {100 * no_correct / no_testing}')
# The accuracy is 100%. You should be proud of yourself. ;)
```

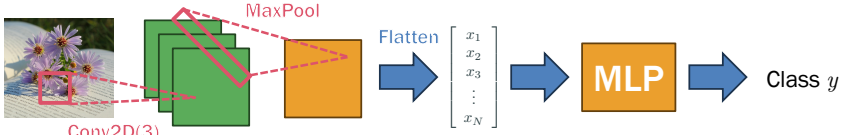
Some Types of Deep Learning Models/1



Unidirectional Recurrent Neural Networks (Uni-RNNs)



Bidirectional Recurrent Neural Networks (Bi-RNNs)



Convolutional Neural Networks (CNNs)

Figure 25: Any deep learning model can be made hybrid by substituting its activation function with a more flexible variational layer. Its rotation gates automatically learn non-linear class boundaries.

Some Types of Deep Learning Models/2

- Transformer model is the foundation of modern-day generative AI, e.g. LLMs

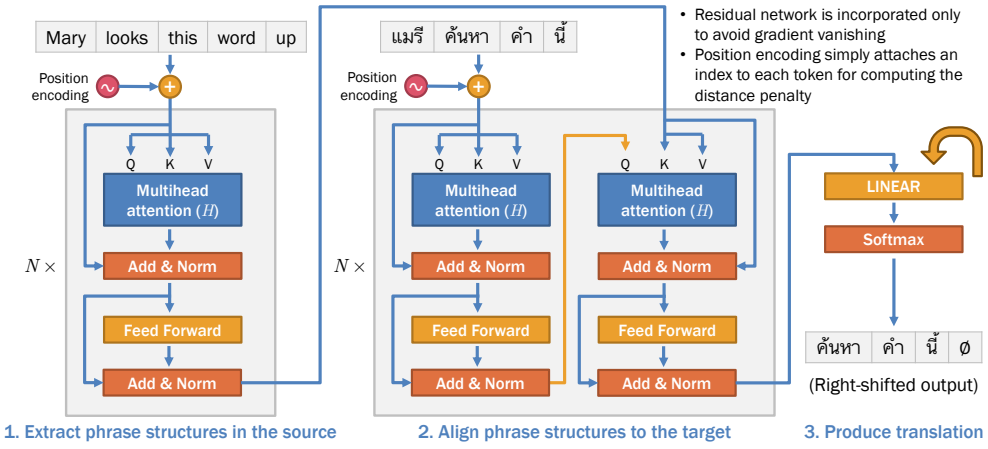


Figure 26: Transformer model can also be made hybrid in the same fashion.

Limitations

- Barren plateau (pl. plateaux [FR])
  - In a high-dimensional parameter space, the prediction loss's landscape is mostly flat, resulting in very small gradients
  - Very small gradients cannot determine the optimal direction for the next moves
  - The SGD algorithm becomes inefficient
- Scalability
  - Interaction between a classical computer and a quantum computer is prone to communication latency
  - Variational quantum algorithms usually have a significantly larger number of parameters, requiring more qubits than the other quantum algorithms

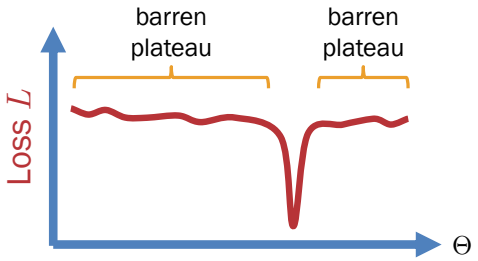


Figure 27: Barren plateau is a flat area of prediction loss in the parameter space. Gradients computed in this area are generally very small. This makes the update equation of the SGD algorithm struggles to reach an optimal point.

Exercise 8.4: Quantum Neural Networks

Answer the following questions.

1. Discuss the differences between classical perceptron and quantum perceptron.
2. Recall that there are three data embedding methods: basic embedding, angular embedding, and amplitude embedding. Regarding the data encoding of quantum perceptron, is it possible to substitude angular embedding with amplitude embedding? Discuss challenges that we may have to encounter if we do so.
3. Explain why we do NOT need any activation function in the quantum perceptron.
4. Study the PennyLane implementation of an XOR gate again. Observe that the input layer is 2-dimension, the hidden layer is 4-dimension, and the output layer is 1-dimension. Are we doing dimensionality reduction or expansion? Why?
5. In connection with the previous question, how many basic entangler layers are there? How many parameters are there in each layer?
6. Is it possible to perform dimensionality reduction or expansion directly in a quantum circuit? Explain your method.

Exercise 8.4: Quantum Neural Networks (cont'd)

7. Explain how the backpropagation works on a hybrid deep learning model. Analyze the differences in treating classical layers vs. quantum layers with the update equation.
8. Explain how the issue of barren plateaux adversely affects the stochastic gradient descent algorithm.
9. Let's consider Transformer-based large language models, which consist of more than 1 billion parameters. Explain why replacing all of their activation functions with quantum layers will make the training process very slow.
10. Modify the implementation of the XOR model by varying the following factors:
    - Basic entangler layer of 4 dimensions *vs.* strong entangler layer (with 3 inner layers) of 4 dimensions *vs.* 3 basic entanglers of 4 dimensions
    - Batch sizes of 20 items *vs.* 10 items *vs.* 40 items
    - Adam optimizer *vs.* SGD optimizer *vs.* AdamW optimizer

    Then compare these configurations in three aspects:
    - Training iterations
    - Training time
    - Total number of parameters

Conclusion

Conclusion

- Variational quantum algorithms (VQA) are quantum algorithms with adjustable parameters, in which variation in the parameters yields temporary solutions to a problem
- Quantum embedding is a means to convert a data item in classical computing into an equivalent quantum state for VQAs: basic, angular, and amplitude embedding
- An ansatz is a VQA whose parameters represent a problem of interest, and problem-agnostic ansatzes are classified into basic and strong entangler layers
- In hybrid classical-quantum machine learning, we can train a VQA with a classical optimization method (e.g. stochastic gradient descent), where gradients are cascadingly computed with the parameter shift rule method
- We can incorporate a VQA to the deep learning framework, where VQA performs as an adaptive activation function
- Hybrid ML is limited by two issues: the barren plateau issue and the scalability

Questions?

Target Learning Outcomes/1

- Variational Quantum Algorithms (VQA)
  1. Define the structure of a Variational Quantum Algorithm as a hybrid classical-quantum approach to optimization.
  2. Compare and contrast quantum embedding techniques, including basis, angular, and amplitude encoding.
  3. Analyze ansatz architectures (HEA, TEN, and ALT) based on their trade-offs between expressivity, trainability, and hardware efficiency.
- Hybrid Classical-Quantum Machine Learning
  1. Formulate prediction loss functions for quantum states using Mean Squared Error (MSE), Negative Log-Likelihood (NLL), and Hinge Loss.
  2. Apply the Parameter Shift Rule (PSR) to compute gradients of quantum circuits for optimization.
  3. Implement a hybrid model in PennyLane that integrates quantum layers into classical machine learning frameworks like PyTorch.

Target Learning Outcomes/2

- Quantum Kernel Methods
  1. Describe how kernel functions and feature maps enable non-linear class separation by moving data into a high-dimensional feature space.
  2. Architect a quantum support vector machine (QSVM) by integrating a quantum feature map with trainable entangler layers.
  3. Analyze the time complexity and quantum advantage of QSVMs compared to classical SVM implementations.
- Quantum Neural Networks (QNN)
  1. Differentiate between classical perceptrons and quantum perceptrons (Qurons), specifically regarding the role of activation functions.
  2. Evaluate the limitations of training deep quantum networks, such as the Barren Plateau problem and communication latency.