Doctorate of Philosophy

Schrödinger's Catwalk

Brian Flynn

University of Bristol

March, 2021

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LISTINGS

ACRONYMS

C	carbon
^{14}N	nitrogen-14
AI	artificial intelligence
AIC	Akaike information criterion
AICC	Akaike information criterion corrected
BF	Bayes factor
BFEER	Bayes factor enhanced Elo ratings
BIC	Bayesian information criterion
CLE	classical likelihood estimation
CPU	central processing unit
DAG	directed acyclic graph
EDH	experiment design heuristic
ES	exploration strategy
ET	exploration tree
FH	Fermi-Hubbard
FN	false negatives
FP	false positives
GA	genetic algorithm
GES	genetic exploration strategy
GPU	graphics processing unit
HPD	high particle density
IQLE	interactive quantum likelihood estimation
JWT	Jordan Wigner transformation
LE	Loschmidt echo

LTL log total likelihood

ML machine learning

MVEE minimum volume enclosing ellipsoid
MW microwave

NN neural network
NV nitrogen-vacancy
NVC nitrogen-vacancy centre

OF objective function

PBS portable batch system
PGH particle guess heuristic
PL photoluminescence

QC quantum computer
QHL quantum Hamiltonian learning
QL quadratic loss
QLE quantum likelihood estimation
QM quantum mechanics
QML quantum machine learning
QMLA Quantum Model Learning Agent

SMC sequential monte carlo
SVM support vector machine

TLTL total log total likelihood
TN true negatives
TP true positives

VQE variational quantum eigensolver

GLOSSARY

Q	Quantum system which is the target of Quantum Model Learning Agent, i.e. the system to be characterised
champion model	The model deemed by Quantum Model Learning Agent (QMLA) as the most suitable for describing the target system
chromosome	A single candidate, in the space of valid solutions to the posed problem in a genetic algorithm
expectation value	Average outcome expected by measuring an observable of a quantum system many times, **??**
gene	Individual element within a chromosome
hyperparameter	Variable within an algorithm that determines how the algorithm itself proceeds
instance	A single implementation of the QMLA algorithm, resulting in a nominated champion model
likelihood	Value that represents how likely a hypothesis is. Usually used in the context of likelihood estiamation, **??**
model	The mathematical description of some quantum system, **??**
model space	Abstract space containing all descriptions (within defined constraints such as dimension) of the system as models
probe	Input probe state, $\|\psi\rangle$, which the target system is initialised to, before unitary evolution
results directory	Directory to which the data and analysis for a given run of QMLA are stored
run	Collection of QMLA instances, usually targeting the same system with the same initial conditions

spawn	Process by which new models are generated, ususally by combining previously considered models
success rate	Fraction of instances within a run where QMLA nominates the true model as champion
term	Individual constituent of a model, e.g. a single operator within a sum of operators, which in total describe a Hamiltonian.
volume	Volume of a parameter distribution's credible region, **??**
win rate	For a given candidate model, the fraction of instances within a run which nominated it as champion

1 MACHINE LEARNING

Machine learning (ML) is the application of statistics, algorithms and computing power to discover meaning and/or devise actions from data. ML has become an umbrella term, encompassing the family of algorithms which aim to leverage computers to learn without being explicitly programmed, as opposed to the more general artificial intelligence (AI), which seeks to make computers behave intelligently, admitting explicit programs to achieve tasks [1]. Its history is therefore imprecise since a number of early, apparently unrelated algorithms were proposed independently, which now constitute ML routines [2, 3]. Nevertheless, the field of ML has been advancing rapidly since the second half of the 20$^{\text{th}}$ century [4], especially recently due to the availability of advanced hardware such as graphics processing units (GPUs), facilitating significant progress through an ever-increasing aresnal of powerful open source software [5, 6, 7].

Throughout this thesis, we endeavour to combine known methods from the ML literature with capabilities of quantum computers (QCs)[1]. Typical ML algorithms, which rely on central processing units (CPUs) or GPUs, are deemed *classical* machine learning, in contrast with quantum machine learning (QML), where QCs are central to processing the data. Similarly to the remit of **??**, here we do not provide an exhaustive account of ML algorithms; we describe only the concepts which are used in later chapters, referring readers to standard texts for a wider discussion [4, 8].

1.1 CLASSICAL MACHINE LEARNING

The first step in any ML application is to consider the types of available data, with respect to the ensemble of known algorithms. Classical ML is usually described in three categories, broadly based on the format of data on which the insight can be built; we will briefly describe each to provide context to discussions throughout this thesis. Later in this thesis we will use the word *model* for descriptions of quantum systems, but here *model* refers to the mapping between inputs and outputs, devised by the ML algorithm.

1.1.1 *Supervised machine learning*

Models are trained using *labelled* data, i.e. each training sample has a known label y_i – or, a set of *feature vectors* $\{\vec{x}_i\}$ are associated with the set of corresponding *classes* $\{y_i\}$ [9]. The output is a predictive tool which aims to reconstruct the classes of unseen feature vectors: in general, we can view the role of ML in this setting as distilling the function f such that

$$f : \vec{x}_i \longmapsto y_i \quad \forall(\vec{x}_i, y_i). \tag{1.1}$$

1 Or simulated QCs, in this thesis.

There are a number of families of algorithms even within the broad category of supervised ML, we define them as follows.

CLASSIFICATION algorithms aim to produce models which can assign unseen instances to the most appropriate label, from a fixed set of available labels [10].

0.1. e.g. labels indicate animals' species, and the feature vector for each sample (data point) encodes the animals' height, weight, number of legs, etc.

REGRESSION models capture the formulaic relationship – which can be either linear or polynamial – between numerical features a target scalar value, by determining coefficients for each feature.

0.2. e.g. y is the salary of employees in a company, and the feature vector consists of the individual employees' age, seniority, experience in years, etc.

NEURAL NETWORKS *Universal function approximators*[2]. By invoking a set of linear and non-linear transformations on input data, the *network* is a function of some paramterisation w; neural networks aim to find the optimal network, w', such that Eq. (1.1) is satisfied, $f(w') : \vec{x}_i \longmapsto y_i$.

0.3. Usually used for classification.

0.3.1. e.g. input *neurons*[3] encode the pixel values of images. The neural network can be used to classify the objects it detects within the image.

SUPPORT VECTOR MACHINES Distinguish similar data points by projecitng data into higher dimensional space, and therein finding the hypersurface which separates classes [12].

0.4. Usually used for classification tasks.

0.5. For data $\mathcal{D} \in \mathbb{R}^n$, a hypersurface in $\mathbb{R}^{n-1}$, can be drawn arbitrarilily through the space (e.g. a 2D plane in a 3D space).

0.6. Unseen data can then classified by which the partition they reside in when projected into the n-dimensional space.

0.7. The task of the support vector machines (SVMs) is to orient the hypersurface in such a way as to separate distinct classes.

Supervised ML algorithms rely on the existence of a body of labelled samples – the dataset $\mathcal{D}$ – upon which the model can be trained. Training is typically performed on a subset of the data, $\mathcal{D}_t$, usually 80% of samples chosen at random. The remaining (20%) of samples, $\mathcal{D}_v$, are retained for the evaluation of the resultant model: $d \in \mathcal{D}_v$ are not trained upon, so do not contribute to

2 Including deep learning networks [11].

3 The term *neuron*, also known as *node* or *unit*, derives from the motivation for this class of algorithm: the cells in the brain used for processing information.

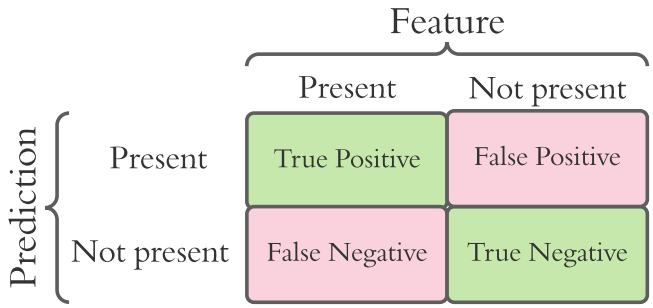


Figure 1.1: Confusion matrix showing the meaning of true positives, true negatives, false positives and false negatives.

the structure of f as returned by the algorithm. The model therefore can not *overfit*, i.e. simply recognise particular samples and label them correctly, without any meaningful inference. The evalutation thus captures how the model can be expected to perform on future, unlabelled data.

1.1.2 *Performance metrics*

The fair assessment of algorithms is achieved through a number of *performance metrics*. In each supervised ML algorithm, the machine attempts to learn the structure of f that optimises some internal objective function (OF), e.g. minimising the average distance between predicted and target labels for regression, $\sum_{d \in \mathcal{D}_t} y_d - y'_d$. To assess the resultant model, we introduce a number of *performance metrics*, which aim to measure serveral perspectives of the model's efficacy, by considering the model's predictions with respect to $\mathcal{D}_v$.

By definition of the data format, it is relatively straightforward to define metrics for supervised routines: the classes assigned to feature vectors, y'_i, can be quantitatively assessed with respect to their true class, y_i. For example, in *binary classification*, the output of the model is either correct or incorrect, allowing us to meaningfully assess its average performance. Likewise, for numerical targets, the difference $\left|y_i - y'_i\right|$ for each sample cumulatively indicate the strength of the model.

There are a large number of quantities and performance metrics against which to judge models' outputs. In binary classification, we care about whether the model predicts that a given feature is present, and whether it predicts features incorrectly. For example, the model aims to classify whether or not a dog is present in an image. These type of binary predictions have four outcomes, which can be summarised in a *confusion matrix* (Fig. 1.1), defined as in Table 1.1.

	y has feature	y' has feature
True positives (TP)	✓	✓
False positives (FP)	×	✓
True negatives (TN)	×	×
False negatives (FN)	✓	×

Table 1.1: Classification metrics. We define classification outcomes based on whether the considered feature was present and/or predicted.

We can use the concepts of Table 1.1 to define a series of *rates* which characterise the model's predictions.

ACCURACY The overall rate at which the algorithm predicts the correct result.

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FN} + \text{FP}} \tag{1.2a}$$

PRECISION Positive predicitve rate. Of those *predicted to have* the feature of interest, what percentage *actually have* the feature.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{1.2b}$$

SENSITIVITY True positive rate (also known as *recall*). Of those which *actually have* the feature, what percentage are *predicted to have* the feature.

$$\text{sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{1.2c}$$

SPECIFICITY True negative rate. Of those which *do not actually have* the feature of interest, how many are *predicted not to* have the feature.

$$\text{specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}} \tag{1.2d}$$

Each metric has clear advantages, but consider also their drawbacks:

- Accuracy can be extremely misleading. For example, consider in a dataset of 10,000 samples, of which only 100 contain the feature of interest. A binary model which predicts every instance as False will achieve $\text{TN} = 9,900, \text{FN} = 100$, receiving an overall accuracy $=$ 99%, despite not having found a single positive sample. This is clearly not useful in identifying the minority of cases of actual interest.
- Sensitivity can be inflated by over-fitting to positive cases. That is, by predicting the feature as present in all cases, all true instances of the feature will be found, however all False

True positives	False negatives	False positives	Precision	Sensitivity	F_1-score
500	500	1000	33	50	$(\frac{2\times33\times50}{33+50}) = 37$
500	500	500	50	50	$(\frac{2\times50\times50}{50+50}) = 50$
1000	0	1000	50	100	$(\frac{2\times50\times100}{50+100}) = 67$
1000	0	0	100	100	$(\frac{2\times100\times100}{100+100}) = 100$

Table 1.2: Examples of how F_1-scorebehaves for varying true positives, false negatives and false positives.

instances will be labelled as having the feature, so the model has not helped separate the data. The model willl yield a high rate of true positives (TP) but also a high rate of false positives (FP).

- Precision can be high for extremely selective models, i.e. those which are conservative in predicting the presence of the feature. By predicting relatively few positive instances, it can ensure that a high proportion of its predictions are correct. The absolute number of instances identified, however, is relatively low as a proportion of the total number in the dataset.
- Specificity, similar to sensitivity, can easily mislead by identifying very few instances as having the target feature. Then, it will correctly predict most non-present instances as False, but will not identify the few instances of interest.

Clearly, the performance metric must be chosen with due consideration for the application; e.g. in testing for a medical condition, rather than incorrectly telling a patient they do not have the condition because the model predicted they were feature-negative, it is preferable to incorrectly identify some patients as feature-positive (since they can be retested). In this case, high accuracy is crucial, at the expense of precision. It is appropriate to blend together these metrics, in order to derive performance metrics which balance the priorities of the outcomes. In general – including in this thesis – the most important aspects of a ML are precision and sensitivity: a model which performs well with respect to *both* of these is sensitive to the feature, but precise in its predictions. A quantity which captures both of these is the F_β-score,

$$f_\beta = \left(1+\beta^2\right) \frac{\text{precision} \times \text{sensitivity}}{(\beta^2 \times \text{precision}) + \text{sensitivity}}, \tag{1.3}$$

where $\beta \in \mathbb{R}$ is the relative weight of priority of sensitivity with respect to precision precision. In particular, considering precision and sensitivity as equally important, i.e. $\beta = 1$, we have the F_1-score,

$$f_1 = \frac{2 \times \text{precision} \times \text{sensitivity}}{\text{precision} + \text{sensitivity}}. \tag{1.4}$$

For examples of how F_1-score balances these considerations, see Table 1.2.

1.1.3 *Unsupervised machine learning*

Contrary to supervised algorithms, unsupervised methods operate on *unlabelled* data, $\mathcal{D}$. This is often summarised as finding structure within unstructured data. Although we do not utilise these methods in this thesis, we briefly summarise them here for completeness; again, we can further compartmentalise methods under this umbrella as follows [13].

CLUSTERING Finding datapoints which are similar to each other, according to some distance metric.

- e.g. online retailers grouping together customers with similar preferences, in order to tailor advertising campaigns.

DIMENSIONALITY REDUCTION Reducing the feature vector of each sample in a dataset to essential components, which may be amalgamations of original features, while retaining structure within the data.

- This can be used for visualisation to allow for inspection of complex datasets, e.g. plotting users of a social network as nodes on a 2D map, where distinct social groups are kept distant.

ASSOCIATION LEARNING Discover correlations among data.

- For instance, a supermarket may find that purchasers of certain products are likely also to buy others, providing actionable insight. For example, purchases involving bread also include butter in 50% of cases, so positioning these nearby may increase sales by reminding consumers of their compatibility.

SEMI-SUPERVISED LEARNING Combine elements of supervised and unsupervised algorithms, to achieve a task beyond the remit of either alone. This often means classifying data where only occur a small subset of the total dataset is labelled.

- e.g. in facial recognition, a clustering algorithm finds similarities between individual people in photos, and identifies a single person present across a number of photos, and associates those photos together. It combines this with a small set of photos for which people have been tagged, locates the same person and automatically tags them in the wider set of photos.

1.1.4 *Reinforcement learning*

A third category of ML algorithms are reinforcement learning (RL). These are methods where an *agent* interacts with some environment, and refines a *policy* for reacting to different stimuli. As such, the agent can, in principle, deal with a wide array of situations. These methods

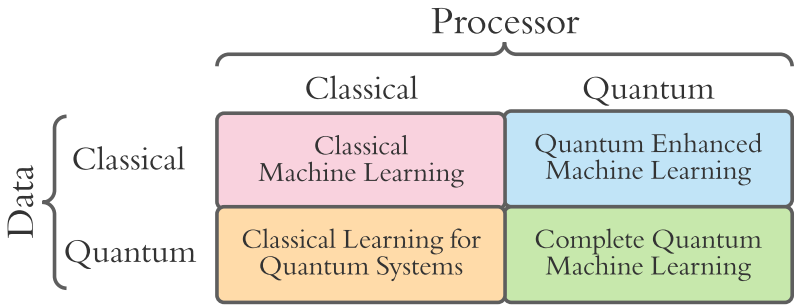


Figure 1.2: Types of quantum machine learning.

underly technologies such as self-driving cars, which inspect their surroundings through *sensors*; compute an *action* according to the policy; implement that action through *actuators*. Following an action, the agent senses whether that action was beneficial or detrimental, and receives a *reward* or *punishment* accordingly. Highly rewarded actions are likely to be repeated in future, allowing the agent to learn what response is appropriate to situational parameters, e.g. a self-driving car braking at a red light is rewarded, while braking at a green light is punished.

The concept of a machine's *agency* is an ongoing discussion in the ML community [14, 15], and will be important in this thesis when we define our model learning protocol as an agent, we will revisit the concept in **??**.

1.2 QUANTUM MACHINE LEARNING

A growing domain is the development of ML algorithms which run on quantum hardware, or exploit data from quantum systems; generally referred to as quantum machine learning (QML) [16, 17]. There are a number of methodologies which are referred to as QML; for clarity, we define the main branches here, as shown in Fig. 1.2.

CLASSICAL MACHINE LEARNING refers to standard ML as described in Section 1.1, i.e. where the processors are CPUs or GPUs, and the applications are not of specific interest to problems in the quantum domain. Recently, this branch has encompassed quantum *inspired* ML, which still target classical problems, but use subroutines which were orginally found in the context of QML [18].

QUANTUM ENHANCED MACHINE LEARNING A quantum co-processor is leveraged on classical data for some provable speedup, i.e. data that could otherwise be processed purely classically, is encoded, loaded onto and processed by quantum hardware. The quantum counterparts of classical ML algorithms aim to solve the same problems, e.g. as neural networks (NNs) [19, 20] and principle component analysis [21].

classical learning for quantum systems Classical processors are employed to extract insight on problems arising from quantum systems, e.g. data is taken from a quantum system and analysed via purely classical methods. For instance, methods which aim to represent quantum states efficiently by leveraging NNs [22, 23].

complete quantum machine learning Data of a quantum nature is processed – at least partially – by quantum processors. The most common technique here is variational quantum eigensolver (VQE), which simulates quanutm systems on QCs, in order to retrieve quantum systems' ground states [24]. The algorithm relies on a classical optimisation routine, but was devised explicitly for implementation on quantum hardware.

The algorithms described in **??** and the applications in **??**–**??** can be described as classical learning for quantum systems. This is because the data upon which the applications are built represent quantum systems, but are processed through classical ML algorithms in order to derive insight about those systems. We caveat that it is feasible, and indeed the long term intention of such algorithms, to run in conjunction with a quantum co-processor, which would represent and evolve quantum systems, but all processing presented here are through strictly classical architecture.

1.3 Genetic algorithms

In later chapters (**??** and **??**) we will use a class of optimisation techniques known as *evolutionary algorithms* [25, 26]. In particular, *genetic algorithms (GAs)* are central to our primary applications. Here we describe genetic algorithms (GAs) in general terms for reference throughout.

GAs work by assuming a given problem can be optimised, if not solved, by a single candidate among a fixed, closed space of candidates, called the population, $\mathcal{P}$. A number of candidates are sampled at random from $\mathcal{P}$ into a single *generation*, and evaluated through some objective function (OF), which assesses the fitness of the candidates at solving the problem of interest. Candidates from the generation are then mixed together to produce the next generation's candidates: this *crossover* process aims to combine only relatively strong candidates, such that the average candidates' fitness improve at each successive generation, mimicing the biological mechanism whereby the genetic makeup of *offspring* is an even mixture of both parents through the philosophy of *survival of the fittest*. The selection of strong candidates as parents for future generations is therefore imperative; in general parents are chosen according to their fitness as determined by the OF. Buidling on this biological motivation, much of the power of GAs comes from the concept of *mutation*: while offspring retain most of the genetic expressions of their parents, some elements are mutated at random. Mutation is crucial in avoiding local optima of the OF landscape by maintaining diversity in the examined subspace of the population.

GAs are not defined either as supervised or unspervised methods; this designation depends on the OF. If candidates are evaluated with respect to labelled data, we can consider that GA

supervised, otherwise unsupervised. Pseudocode for a generic GA is given in Algorithm 1, but we can informally define the procedure as follows. Given access to the population, $\mathcal{P}$,

1. Sample N_m candidates from $\mathcal{P}$ at random
   (a) call this group of candidates the first generation, μ.
2. Evaluate each candidate $\gamma_j \in \mu$
   (a) each γ_j is assigned a fitness, g_j;
   (b) the fitness is computed through an objective function acting on the candidate, i.e. $g_j = g(\gamma_j)$.
3. Map the fitnesses of each candidate, $\{g_j\}$, to selection probabilities for each candidate, $\{s_j\}$
   (a) e.g. by normalising the fitnesses, or by removing some poorly-performing candidates and then normalising.
4. Generate the next generation of candidates
   (a) Reset $\mu = \{\}$;
   (b) Select pairs of parents, $\{\gamma_{p_1}, \gamma_{p_2}\}$, from μ
      i. Each candidate's probability of being chosen is given by their s_j.
   (c) Cross over $\{\gamma_{p_1}, \gamma_{p_2}\}$ to produce children candidates, $\{\gamma_{c_1}, \gamma_{c_2}\}$
      i. mutate $\gamma_{c_1}, \gamma_{c_2}$ according to some random probabilistic process;
      ii. keep γ_{c_i} only if it is not already in μ, to ensure N_m unique candidates are tested at each generation.
   (d) until $|\mu| = N_m$, iterate to step (b).
5. Until the N_g^{th} generation is reached, iterate to step 2.;
6. The strongest candidate on the final generation is deemed the solution to the posed problem.

Candidates are manifested as *chromosomes*, i.e. strings of fixed length, whose entries, called *genes*, each represent some element of the system. In general, genes can have continuous values, although usually, and for all purposes in this thesis, genes are binary, capturing simply whether or not the gene's corresponding feature is present in the chromosome.

1.3.1 *Example: knapsack problem*

One commonly referenced combinatorial optimisation problem is the *knacksack problem*, which we will use to elucidate the abstract concepts described so far. The knapsack problem is stated as: given a set of objects, where each object has a defined weight and value, determine the set of objects to pack in a knapsack which can support a limited weight, such that the value of the packed objects is maximised. Say there are n objects; we can write the vector containing the values of those objects as $\vec{v}$, and the vector of their weights as $\vec{w}$. We can then represent

Algorithm 1: Genetic algorithm

Input: $\mathcal{P}$ // Population of candidate solutions to given problem
Input: $g()$ // objective funtion
Input: map_g_to_s() // function to map fitness to selection probability
Input: select_parents() // function to select parents among generation
Input: crossover() // function to cross over two parents to produce offspring
Input: mutate() // function to mutate offspring probabilistically
Input: N_g // number of generations
Input: N_m // number of candidates per generation

Output: γ' // strongest candidate

$\mu \leftarrow \text{sample}\,(\mathcal{P}, N_m)$
for $i \in 1, ..., N_g$ **do**
  for $\gamma_j \in \mu$ **do**
    $g_j \leftarrow g(\gamma_j)$ // assess fitness of candidate
  end
  $\{s_j\} \leftarrow$ map_g_to_s($\{g_j\}$) // map fitnesses to normalised selection probability
  $\mu_c = \arg\max_{s_j}\{\gamma_j\}$ // record champion of this generation

  $\mu \leftarrow \{\}$ // empty set for next generation
  while $|\mu| < N_m$ **do**
    $p_1, p_2 \leftarrow$ select_parents($\{s_j\}$) // choose parents based on candidates' s_j
    $c_1, c_2 \leftarrow$ crossover(p_1, p_2) // generate offspring candidates based on parents
    $c_1, c_2 \leftarrow$ mutate(c_1, c_2) // possibly mutate offspring
    for $c \in \{c_1, c_2\}$ **do**
      if $c \notin \mu$ **then**
        $\mu \leftarrow \mu \cup \{c\}$ // keep if child is new
      end
    end
  end
end
$\gamma' \leftarrow \arg\max_{s_j}\{\gamma_j \in \mu\}$ // strongest candidate on final generation

return γ'

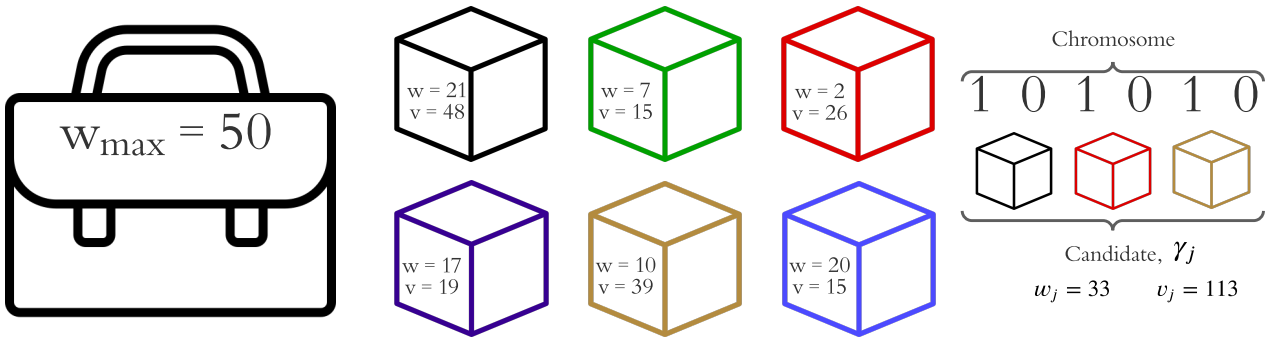


Figure 1.3: Depiction of the knapsack problem. **Left**, A knapsack which can hold any number of objects but is constrained by the total weight it can support, $w_{max} = 50$. **Centre**, A set of objects are available, each with associated weight, w, and value v. The objective is to find the subset of objects which maximise the total value, while not exceeding the capacity of the knapsack. **Right**, An example chromosome, i.e. candidate γ_j, where the bits of the chromosome indicate whether the corresponding object is included, allowing for calculation of the total weight and value of the candidate solution, w_j, v_j.

configurations of objects – i.e. candidate solutions to the problem – as vectors $\vec{\gamma}_j$, whose elements are binary, and simply indicate whether or not the associated object is included in the set. The candidate vector $\vec{gamma}_j$ is equivalent to the chromosome γ_j. For example, with $n = 6$,

$$\vec{\gamma}_j = (1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1) \implies \gamma_j = 100001 \tag{1.5}$$

indicates a set of objects consisting only of those indexed first and last, with none of the intermediate objects included.

The fitness of any candidate is then given by the total value of that configuration of objects, $v_j = \vec{v} \cdot \vec{\gamma}_j$, but candidates are only admitted[4] if the weight of the corresponding set of objects is less than the capacity of the knapsack, i.e. $w_j = \vec{w} \cdot \vec{\gamma}_j \leq w_{max}$.

For example where each individual object has value < 50 and weight < 25 and $w_{max} = 50$, recalling $\gamma_j = 100001$, say,

$$\vec{v} = (48 \quad 15 \quad 26 \quad 19 \quad 39 \quad 15) \implies v_j = \vec{\gamma}_j \cdot \vec{v} = 48 + 15 = 63; \tag{1.6a}$$

$$\vec{w} = (21 \quad 7 \quad 2 \quad 17 \quad 10 \quad 20) \implies w_j = \vec{\gamma}_j \cdot \vec{w} = 21 + 20 = 41. \tag{1.6b}$$

We can hence assess the fitness of γ_j as 63 and deem it a valid candidate since it does not exceed the weight threshold. We can likewise compute the total weight and value of a series

4 Note there are alternative strategies to dealing with candidates who violate the weight condition, such as to impose a penalty within the OF, but for our purposes let us assume we simply disregard violators.

Name	Candidate	Value	Weight	Valid
γ_1	110011	117	58	No
γ_2	101010	113	33	Yes
γ_3	011110	99	36	Yes
γ_4	011011	95	39	Yes
γ_5	111000	89	30	Yes
γ_6	010111	88	54	No
γ_7	100010	87	31	Yes
γ_8	110001	78	48	Yes
γ_9	011101	75	46	Yes
γ_{10}	110000	63	28	Yes
γ_{11}	000011	54	30	Yes
γ_{12}	000101	34	37	Yes

Table 1.3: Candidate solutions to the knapsack problem for randomly generated chromosomes.

of randomly generated candidates, and deem them valid or not. Table 1.3 shows a set of 12 randomly generated candidates, of which ten are valid.

The strongest (valid) candidates from Table 1.3 are 101010, 011110. By spawning from these candidates through a one-point crossover at the midpoint[5], we get $\gamma_{c_1} = 101110, \gamma_{c_2} = 011010$, from which we can see $v_{c_1} = 132, w_{c_1} = 50$, i.e. by combining two strong candidates we produce the strongest-yet-seen valid candidate.

By repeating this procedure, it is expected to uncover candidates which optimise v_j while maintining $w_j \leq w_{max}$, or at least to produce near-optimal solutions, using far less time/resources than brute-force evaluation of all candidates, which is usually sufficient. For instance, with $n = 100$ objects, there are $2^{100} \approx 10^{30}$ candidates to consider; the most powerful supercomputers in the world currently claim on the order of Exa-FLOPs, i.e. 10^{18} operations per second, of which say $\mathcal{O}(1000)$ operations are required to test each candidate, meaning 10^{15} candidates can be checked per second in a generous example. This would still require 10^{12} seconds to solve absolutely, so it is reasonable in cases like this to accept *approximately optimal* solutions[6].

1.3.2 *Selection mechanism*

A key subroutine of every GA is the mechanism through which it nominates candidates from generation μ as parents to offsping candidates in $\mu + 1$ [27]. All mechanisms have in common

5 One-point crossovers are detailed in Section 1.3.3 with this example shown in Fig. 1.5.

6 Simply put: in machine learning, *good enough* is good enough. We will adopt this philosophy for the remainder of this thesis and life.

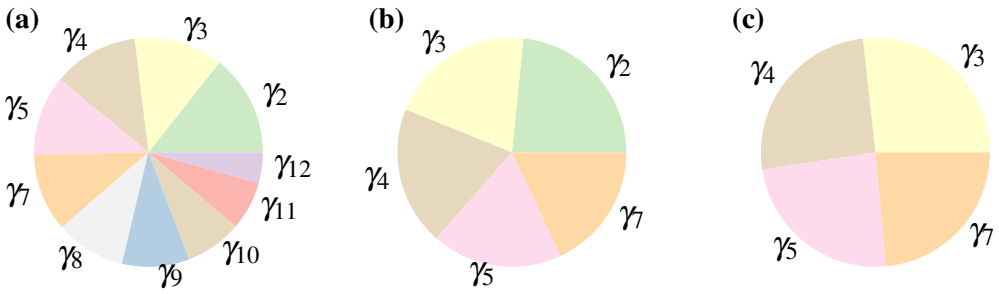


Figure 1.4: Roulette wheels showing selection probability s_i for corresponding candidates γ_i. Colours here only distinguish candidates, they do not encode any information. **a**, All valid candidates are assigned selection probability based on their value in Table 1.3. **b**, The set of potential parents is truncated to include only the strongest five candidates. **c**, After one parent (γ_2) has been chosen, it is removed from the roulette wheel and the remaining candidates' probabilities are renormalised for the selection of the second parent.

that they act on a set of candidates from the previous generation, where each candidate, γ_j, has been evaluated and has fitness value, g_j. Among the viable schemes for selecting individual parents from μ are

- Rank selection: candidates are selected with probabilty proportional to their ranking relative to the fitness of contemporary candidates in the same generation;
- Tournament selection: a subset of k candidates are chosen at random from μ, of which the candidate with the highest fitness is taken as the parent;
- Stochastic universal sampling: candidates are sampled proportional to their fitness, but the sampling algorithm is biased to ensure high-fitness candidates are chosen at least once within the generation.

We will only detail the mechanism used in later applications within this thesis: fitness proportional selection, known as *roulette selection* [27]. This is a straightforward strategy where we directly map candidates' fitness, g_i to a selection probability, s_i, simply by normalising $\{g_i\}$, allowing us to visualise a roulette wheel of uneven wedges, each of which correspond to a candidate. Then we need only conceptually spin the roulette wheel to select the first parent, γ_{p_1}. We then remove γ_{p_1} from the set of potential parents, renormalise the remaining $\{s_i\}$, and spin the wheel again to choose the second parent, γ_{p_2}. The roulette selection is shown in Fig. 1.4.

In practice, we repeat the roulette selection process outlined until the next generation is filled, usually we have $|\mu| = N_m$, and desire that every generation should contain the same N_m candidates, so we repeat the roulette selection $N_m/2$ times per generation, since every pair of parents yields two offspring. It is important that meaningful differences in fitness are reflected by the selection probability, which is difficult to ensure for large N_m, e.g. with 20 candidates, the strongest candidate is only a marginally more probable parent than the worst – this effect is

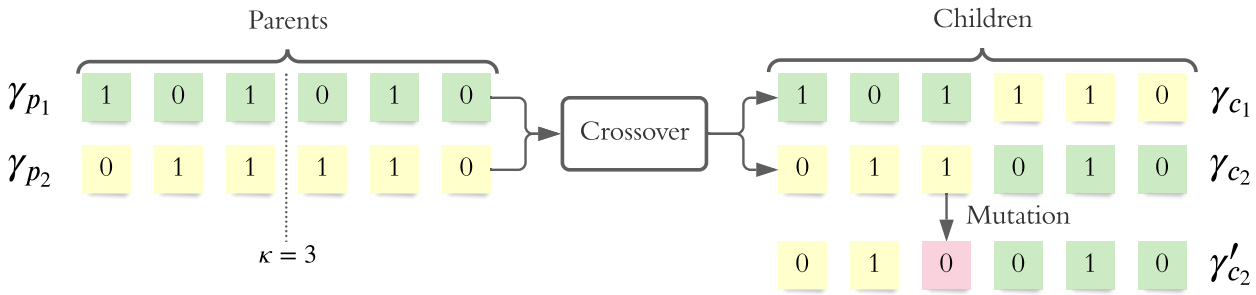


Figure 1.5: Crossover and mutation of chromosomes. Two parents, $\{\gamma_{p_1}, \gamma_{p_2}\}$, are nominated from the process in Fig. 1.4. They are then crossed-over via a one-point crossover with crossing point $\kappa = 3$, resulting in children candidates $\{\gamma_{c_1}, \gamma_{c_2}\}$. One child chromosome is mutated to yield a new candidate, γ'_{c_2}. The candidates added to the next generation are then $\{\gamma_{c_1}, \gamma'_{c_2}\}$.

amplified for larger N_m. We therefore wish to reduce the set of potential parents to ensure high quality offspring: we truncate μ with rate τ to retain only the τN_m highest-fitness candidates as selectable parents.

1.3.3 *Reproduction*

When a pair of parents have been nominated by the selection mechanism above, it remains to use those parents to *reproduce*, i.e. to produce offspring which should inherit and improve upon the properties of their parents. Here we use a *one-point crossover*, whereby the two parent chromosomes are mixed together to form two offspring, about a single point, κ: for candidates of n genes, the first κ genes of γ_{p_1} are conjoined with the latter $n - \kappa$ genes of γ_{p_2}. Often κ is restricted to the midpoint of the chromosomes, although in general it need not be: we will instead consider $\kappa \in \left(\frac{n}{4}, \frac{3n}{4}\right)$, e.g. with $n = 12$, $\kappa \in (3, 9)$. The one-point crossover is shown for $n = 6$ with $\kappa = 3$ in Fig. 1.5, recalling the chromosome structure from Section 1.3.1.

By allowing κ other than the midpoint, we drastically increase the number of combinations of parents available for reproduction. Finally, then, parent selection is done by constructing a database of pairs of potentital parents with all available crossover points, with selection probability given by the product of their individual fitnesses. This is conceptually equivalent to selection via roulette wheel as above. Recalling the fitnesses (values) of Table 1.3, we generate the parent selection database in Table 1.4.

The GA maintains diversity in the subspace of $\mathcal{P}$ it studies, by *mutating* some of the newly proposed offspring candidates. Again, there are a multitude of approaches for this step [28], but for brevity we only describe the one used in this thesis. For each proposed child candidate, γ_c, we probabilistically mutate each gene with some mutation rate r_m: if a mutation occurs, the

Parent 1	Parent 2	κ	s_{ij}
γ_2	γ_3	2	$11,187\ (= 113 \times 99)$
γ_2	γ_3	3	$11,187$
γ_2	γ_3	4	$11,187$
γ_2	γ_4	2	$10,735\ (= 113 \times 95)$
γ_2	γ_4	3	$10,735$
γ_2	γ_4	4	$10,735$
		$\vdots$	
γ_5	γ_7	2	$7,743\ (= 89 \times 87)$
γ_5	γ_7	3	$7,743$
γ_5	γ_7	4	$7,743$

Table 1.4: Example of parent selection database. Pairs of parents are selected together, with the (unnormalised) selection probability, s_{ij}, given by the product of the individual candidates' fitnesses. Pairs of parents are repeated in the database for differing κ, and all κ are equally likely.

child is replaced by γ_c'. That is, γ_c' is added to the next generation, and γ_c is discarded. r_m is a *hyperparameter* of the GA: the performance of the algorithm can be optimised by finding the best r_m for a given problem.

1.3.4 *Candidate evaluation*

Within every generation of the GA, each candidate must be evaluated, so that the relative strength of candidates can be exploited in constructing candidates for the next generation. In the example of the knapsack problem, candidate solutions were evaluated by the value of their contents, but also by whether they would fit in the knapsack. Idenitfiyng the appropriate method by which to evaluate candidates is arguably the most important aspect of designing a GA: while the choice of hyperperameters ($N_g, N_m, \tau.r_m$) dictate the efficacy of the search, the lack of an effective metric by which to distinguish candidates would render the procedure pointless. Considerations are hence usually built into the objective function; GA implementations later in this thesis therefore demand we design objective functions with respect to the individual application.

APPENDIX

A

FIGURE REPRODUCTION

Most of the figures presented in the main text are generated directly by the Quantum Model Learning Agent (QMLA) framework. Here we list the implementation details of each figure so they may be reproduced by ensuring the configuration in Table A.1 are set in the launch script. The default behaviour of QMLA is to generate a results folder uniquely identified by the date and time the run was launched, e.g. results can be found at the *results directory* qmla/Launch/Jan_01/12_34. Given the large number of plots available, ranging from high-level run perspective down to the training of individual models, we introduce a plot_level $\in \{1, ..., 6\}$ for each run of QMLA: higher plot_level informs QMLA to generate more plots.

Within the results directory, the outcome of the run's instances are stored, with analysis plots broadly grouped as

- evaluation: plots of probes and times used as the evaluation dataset.
- instances: outcomes of an individual QMLA instance, grouped by the instance ID. Includes results of training of individual models (in model_training), as well as sub-directories for anlaysis at the branch level (in branches) and comparisons.
- combined_datasets: pandas dataframes containing most of the data used during analysis of the run. Note that data on the individual model/instance level may be discarded so some minor analyses can not be performed offline.
- exploration_strategy_plots plots specifically required by the exploration strategy (ES) at the run level.
- champion_models: analysis of the models deemed champions by at least one instance in the run, e.g. average parameter estimation for a model which wins multiple instances.
- performance: evaluation of the QMLA run, e.g. the win rate of each model and the number of times each term is found in champion models.
- meta analysis of the algorithm' implementation, e.g. timing of jobs on each process in a cluster; generally users need not be concerned with these.

In order to produce the results presented in this thesis, the configurations listed in Table A.1 were input to the launch script. The launch scripts in the QMLA codebase consist of many configuration settings for running QMLA; only the lines in snippet in Listing A.1 need to be set according to altered to retrieve the corresponding figures. Note that the runtime of QMLA grows quite quickly with N_e, N_p (except for the AnalyticalLikelihood ES), especially for the entire QMLA algorithm; running quantum Hamiltonian learning (QHL) is feasible on a personal computer in < 30 minutes for $N_e = 1000; N_p = 3000$.

```
#!/bin/bash

###############
# QMLA run configuration
###############
num_instances=1
run_qhl=1 # perform QHL on known (true) model
run_qhl_mulit_model=0 # perform QHL for defined list of models.
exp=200 # number of experiments
prt=1000 # number of particles

###############
# QMLA settings
###############
plot_level=6
debug_mode=0

###############
# Choose an exploration strategy
###############

exploration_strategy='AnalyticalLikelihood'
```

Listing A.1: QMLA Launch scipt

Figure	Exploration Strategy	N_E	N_P	Data
??	DemoHeuristicPGH	1000	3000	Nov_27/19_39
	DemoHeuristicNineEighths	1000	3000	Nov_27/19_40
	DemoHeuristicTimeList	1000	3000	Nov_27/19_42
	DemoHeuristicRandom	1000	3000	Nov_27/19_47
??	DemoProbesPlus	1000	3000	Nov_27/14_43
	DemoProbesZero	1000	3000	Nov_27/14_45
	DemoProbesTomographic	1000	3000	Nov_27/14_46
	DemoProbes	1000	3000	Nov_27/14_47
??	DemoProbesPlus	1000	3000	Nov_27/14_43
	DemoProbesZero	1000	3000	Nov_27/14_45
	DemoProbesTomographic	1000	3000	Nov_27/14_46
	DemoProbes	1000	3000	Nov_27/14_47
??	AnalyticalLikelihood	500	2000	Nov_16/14_28
??	DemoIsing	500	5000	Nov_18/13_56
??	DemoIsing	1000	5000	Nov_18/13_56
??	DemoIsing	1000	5000	Nov_18/13_56
??	IsingLatticeSet	1000	4000	Nov_19/12_04
	IsingLatticeSet	1000	4000	Nov_19/12_04
	IsingLatticeSet	1000	4000	Nov_19/12_04
??	IsingLatticeSet	1000	4000	Sep_30/22_40
	HeisenbergLatticeSet	1000	4000	Oct_22/20_45
	FermiHubbardLatticeSet	1000	4000	Oct_02/00_09

Table A.1: Implementation details for figures used in the main text. Continued in Table A.2.

Figure	Exploration Strategy	N_E	N_P	Data
??	DemoBayesFactorsByFscore	500	2500	Dec_09/12_29
	DemoFractionalResourcesBayesFactorsByFscore	500	2500	Dec_09/12_31
	DemoBayesFactorsByFscore	1000	5000	Dec_09/12_33
	DemoBayesFactorsByFscoreEloGraphs	500	2500	Dec_09/12_32
??	HeisenbergGeneticXYZ	500	2500	Dec_10/14_40
??	HeisenbergGeneticXYZ	500	2500	Dec_10/14_40
	HeisenbergGeneticXYZ	500	2500	Dec_10/14_40
??	HeisenbergGeneticXYZ	500	2500	Dec_10/16_12
	HeisenbergGeneticXYZ	500	2500	Dec_10/16_12
??	HeisenbergGeneticXYZ	500	2500	Dec_18/20_12
??	NVCentreExperimentalData	1000	3000	2019/Oct_02/18_01
	SimulatedExperimentNVCentre	1000	3000	2019/Oct_02/18_16
??	NVCentreExperimentalData	1000	3000	2019/Oct_02/18_01
??	SimulatedExperimentNVCentre	1000	3000	2019/Oct_02/18_16
??	SimulatedExperimentNVCentre	1000	3000	2019/Oct_02/18_16
	NVCentreExperimentalData	1000	3000	2019/Oct_02/18_01
??	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_09/12_00
	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_09/12_00
??	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_09/12_00
	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_09/12_00
??	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_08/23_58
??	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_08/23_58

Table A.2: [Continued from Table A.1] Implementation details for figures used in the main text.

B

FUNDAMENTALS

There are a number of concepts which are fundamental to any discussion of quantum mechanics (QM), but are likely to be known to most readers, and are therefore cumbersome to include in the main body of the thesis. We include them here for completeness[1].

B.1 LINEAR ALGEBRA

Here we review the language of linear algebra and summarise the basic mathematical techniques used throughout this thesis. We will briefly recall some definitions for reference.

- Notation

Definition of	Representation
Vector (or *ket*)	$\|\psi\rangle$
Dual Vector (or *bra*)	$\langle\psi\|$
Tensor Product	$\|\psi\rangle \otimes \|\phi\rangle$
Complex conjugate	$\|\psi^*\rangle$
Transpose	$\|\psi\rangle^T$
Adjoint	$\|\psi\rangle^\dagger = (\|\psi\rangle^*)^T$

Table B.1: Linear algebra definitions.

The dual vector of a vector (ket) $|\psi\rangle$ is given by $\langle\psi| = |\psi\rangle^\dagger$.

The *adjoint* of a matrix replaces each matrix element with its own complex conjugate, and then switches its columns with rows.

$$M^\dagger = \begin{pmatrix} M_{0,0} & M_{0,1} \\ M_{1,0} & M_{1,1} \end{pmatrix}^\dagger = \begin{pmatrix} M^*_{0,0} & M^*_{0,1} \\ M^*_{1,0} & M^*_{1,1} \end{pmatrix}^T = \begin{pmatrix} M^*_{0,0} & M^*_{1,0} \\ M^*_{0,1} & M^*_{1,1} \end{pmatrix} \tag{B.1}$$

1 Much of this description is reproduced from my undergraduate thesis [29].

The *inner product* of two vectors, $|\psi\rangle = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_n \end{pmatrix}$ and $|\phi\rangle = \begin{pmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_n \end{pmatrix}$ is given by

$$\langle\phi|\psi\rangle = (|\phi\rangle^\dagger)\,|\psi\rangle = (\phi_1^* \ \phi_2^* \ \dots \ \phi_n^*) \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_n \end{pmatrix} = \phi_1^*\psi_1 + \phi_2^*\psi_2 + \cdots + \phi_n^*\psi_n \tag{B.2}$$

$|\psi\rangle_i, |\phi\rangle_i$ are complex numbers, and therefore the above is simply a sum of products of complex numbers. The inner product is often called the *scalar product*, which is in general complex.

B.2 POSTULATES OF QUANTUM MECHANICS

There are numerous statements of the postulates of quantum mechanics. Each version of the statements aims to achieve the same foundation, so we endeavour to explain them in the simplest terms.

1 Every moving particle in a conservative force field has an associated wave-function, $|\psi\rangle$. From this wave-function, it is possible to determine all physical information about the system.

2 All particles have physical properties called observables (denoted Q). In order to determine a value, Q, for a particular observable, there is an associated *operator* $\hat{Q}$, which, when acting on the particles wavefunction, yields the value times the wavefunction. The observable Q is then the eigenvalue of the operator $\hat{Q}$.

$$\hat{Q}\,|\psi\rangle = q|\psi\rangle \tag{B.3}$$

3 Any such operator $\hat{Q}$ is Hermitian

$$\hat{Q}^\dagger = \hat{Q} \tag{B.4}$$

4 The set of eigenfunctions for any operator $\hat{Q}$ forms a complete set of linearly independent functions.

5 For a system with wavefunction $|\psi\rangle$, the expectation value of an observable Q with respect to an operator $\hat{Q}$ is denoted by $\langle q\rangle$ and is given by

$$\langle q\rangle = \langle\psi|\hat{Q}|\psi\rangle \tag{B.5}$$

6 The time evolution of $|\psi\rangle$ is given by the time dependent *Schrodinger Equation*

$$i\hbar\frac{\partial\psi}{\partial t} = \hat{H}\psi, \tag{B.6}$$

where $\hat{H}$ is the system's Hamiltonian.

Using these building blocks, we can begin to construct a language to describe quantum systems.

B.3 STATES

An orhthonormal basis consists of vectors of unit length which do not overlap, e.g. $|x_1\rangle = \begin{pmatrix}1\\0\end{pmatrix}, |x_2\rangle = \begin{pmatrix}0\\1\end{pmatrix} \Rightarrow \langle x_1|x_2\rangle = 0$. In general, if $\{|x\rangle\}$ are the eigenstates of a system, then the system can be written as some state vector, $|\psi\rangle$, in general a superposition over the basis-vectors:

$$|\psi\rangle = \sum_x a_x|x\rangle \tag{B.7a}$$

$$\text{subject to} \quad \sum_x |a_x|^2 = 1, \quad a_x \in \mathbb{C} \tag{B.7b}$$

The *state space* of a physical system (classical or quantum) is then the set of all possible states the system can exist in, i.e the set of all possible values for $|\psi\rangle$ such that Eq. (B.7b) are satisfied.

For example, photons can be polarised horizontally ($\leftrightarrow$) or vertically ($\updownarrow$); take those two conditions as observable states to define the eigenstates of a two-level system, so we can designate the photon as a qubit. Then we can map the two states to a 2-dimensional, x-y plane: a general vector on such a plane can be represented by a vector with coordinates $\begin{pmatrix}x\\y\end{pmatrix}$. These polarisations can then be thought of as standard basis vectors in linear algebra. Denote $\leftrightarrow$ as the eigenstate $|0\rangle$ and $\updownarrow$ as $|1\rangle$

$$|\leftrightarrow\rangle = |0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \qquad \text{A unit vector along x-axis} \tag{B.8a}$$

$$|\updownarrow\rangle = |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \qquad \text{A unit vector along y-axis} \tag{B.8b}$$

Now, in relation to the concept of superposition, we can consider, for example, a photon in an even superposition of the vertical and horizontal polarisations, evenly splitting the two basis vectors. As such, we would require that, upon measurement, it is equally likely that the

photon will *collapse* into the polarised state along x as it is to collapse along y. That is, we want $\Pr(\updownarrow) = \Pr(\leftrightarrow)$ so assign equal modulus amplitudes to the two possibilities:

$$|\psi\rangle = a\,|\leftrightarrow\rangle + b\,|\updownarrow\rangle\,, \quad \text{with} \quad \Pr(\updownarrow) = \Pr(\leftrightarrow) \Rightarrow |a|^2 = |b|^2 \tag{B.9}$$

We consider here a particular case, due to the significance of the resultant basis, where $\leftrightarrow$-polarisation and $\updownarrow$-polarisation have real amplitudes $a, b \in \mathbb{R}$.

$$\begin{aligned} &\Rightarrow a = \pm b \quad \text{but also} \quad |a|^2 + |b|^2 = 1 \\ \Rightarrow \quad &a = \frac{1}{\sqrt{2}} \quad ; \quad b = \pm\frac{1}{\sqrt{2}} \\ &\Rightarrow |\psi\rangle = \frac{1}{\sqrt{2}}\,|\leftrightarrow\rangle \pm \frac{1}{\sqrt{2}}\,|\updownarrow\rangle \\ &\Rightarrow |\psi\rangle = \frac{1}{\sqrt{2}}\,|0\rangle \pm \frac{1}{\sqrt{2}}\,|1\rangle \end{aligned} \tag{B.10}$$

These particular superpositions are of significance:

$$|+\rangle = \frac{1}{\sqrt{2}}\,(|0\rangle + |1\rangle) \tag{B.11a}$$

$$|-\rangle = \frac{1}{\sqrt{2}}\,(|0\rangle - |1\rangle) \tag{B.11b}$$

This is called the Hadamard basis: it is an equally valid vector space as the standard basis which is spanned by $\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}$, as it is simply a rotation of the standard basis.

B.3.1 *Mulitpartite systems*

In reality, we often deal with systems of multiple particles, represented by multiple qubits. Mathematically, we consider the state vector of a system containing n qubits as being the tensor product of the n qubits' individual state vectors[2]. For instance, suppose a 2-qubit system, $|\psi\rangle$ consisting of two independent qubits $|\psi_A\rangle$ and $|\psi_B\rangle$:

$$|\psi\rangle = |\psi_A\rangle\,|\psi_B\rangle = |\psi_A\psi_B\rangle = |\psi_A\rangle \otimes |\psi_B\rangle \tag{B.12}$$

Consider first a simple system of 2 qubits. Measuring in the standard basis, these qubits will have to collapse in to one of the basis states $|0,0\rangle, |0,1\rangle, |1,0\rangle, |1,1\rangle$. Thus, for such a 2-qubit system, we have the general superposition

$$|\psi\rangle = \alpha_{0,0}|0,0\rangle + \alpha_{0,1}|0,1\rangle + \alpha_{1,0}|1,0\rangle + \alpha_{1,1}|1,1\rangle$$

2 We will later discuss entangled states, which can not be described thus.

where $\alpha_{i,j}$ is the amplitude for measuring the system as the state $|i,j\rangle$. This is perfectly analogous to a classical 2-bit system necessarily occupying one of the four possibilities $\{(0,0),(0,1),(1,0),(1,1)\}$.

Hence, for example, if we wanted to concoct a two-qubit system composed of one qubit in the state $|+\rangle$ and one in $|-\rangle$

$$\begin{aligned}
|\psi\rangle &= |+\rangle \otimes |-\rangle \\
|\psi\rangle &= \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \otimes \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right) \\
&= \frac{1}{2}\left[|00\rangle - |01\rangle + |10\rangle - |11\rangle\right] \\
&= \frac{1}{2}\left[\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} - \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} - \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix}\right] \\
&= \frac{1}{2}\left[\begin{pmatrix}1\\0\\0\\0\end{pmatrix} - \begin{pmatrix}0\\1\\0\\0\end{pmatrix} + \begin{pmatrix}0\\0\\1\\0\end{pmatrix} - \begin{pmatrix}0\\0\\0\\1\end{pmatrix}\right]. \\
\Rightarrow |\psi\rangle &= \frac{1}{2}\begin{pmatrix}1\\-1\\-1\\1\end{pmatrix}
\end{aligned} \tag{B.13}$$

That is, the two qubit system – and indeed any two qubit system – is given by a linear combination of the four basis vectors

$$\{|00\rangle, |0,1\rangle, |10\rangle, |11\rangle\} = \left\{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\end{pmatrix}\right\}. \tag{B.14}$$

We can notice that a single qubit system can be described by a linear combination of two basis vectors, and that a two qubit system requires four basis vectors to describe it. In general we can say that an n-qubit system is represented by a linear combination of 2^n basis vectors.

B.3.2 *Registers*

A *register* is generally the name given to an array of controllable quantum systems; here we invoke it to mean a system of multiple qubits, specifically a subset of the total number of

available qubits. For example, a register of ten qubits can be denoted $|x[10]\rangle$, and we can think of the system as a register of six qubits together with a register of three and another register of one qubit.

$$|x[10]\rangle = |x_1[6]\rangle \otimes |x_2[3]\rangle \otimes |x_3[1]\rangle$$

B.4 ENTANGLEMENT

Another unique property of quantum systems is that of *entanglement*: when two or more particles interact in such a way that their individual quantum states can not be described independent of the other particles. A quantum state then exists for the system as a whole instead. Mathematically, we consider such entangled states as those whose state can not be expressed as a tensor product of the states of the individual qubits it's composed of: they are dependent upon the other.

To understand what we mean by this dependence, consider a counter-example. Consider the Bell state,

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right), \tag{B.15}$$

if we measure this state, we expect that it will be observed in either eigenstate $|00\rangle$ or $|11\rangle$, with equal probability due to their amplitudes' equal magnitudes. The bases for this state are simply the standard bases, $|0\rangle$ and $|1\rangle$. Thus, according to our previous definition of systems of multiple qubits, we would say this state can be given as a combination of two states, like Eq. (B.12),

$$\begin{aligned} |\Phi^+\rangle &= |\psi_1\rangle \otimes |\psi_2\rangle \\ &= (a_1|0\rangle + b_1|1\rangle) \otimes (a_2|0\rangle + b_2|1\rangle) \\ &= a_1a_2|00\rangle + a_1b_2|01\rangle + b_1a_2|10\rangle + b_1b_2|11\rangle \end{aligned} \tag{B.16}$$

However we require $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, which would imply $a_1b_2 = 0$ and $b_1a_2 = 0$. These imply that either $a_1 = 0$ or $b_2 = 0$, and also that $b_1 = 0$ or a_2=0, which are obviously invalid since we require that $a_1a_2 = b_1b_2 = \frac{1}{\sqrt{2}}$. Thus, we cannot express $|\Phi^+\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$; this inability to separate the first and second qubits is what we term *entanglement*.

B.5 UNITARY TRANSFORMATIONS

A fundamental concept in quantum mechanics is that of performing transformations on states. *Quantum transformations*, or *quantum operators*, map a quantum state into a new state within the same Hilbert space. There are certain restrictions on a physically possible quantum transformation: in order that U is a valid transformation acting on some superposition $|\psi\rangle = a_1|\psi_1\rangle + a_2|\psi_2\rangle + \ldots a_k|\psi_k\rangle$, U must be linear

$$U(a_1|\psi_1\rangle + a_2|\psi_2\rangle + \ldots a_k|\psi_k\rangle) = a_1(U|\psi_1\rangle) + a_2(U|\psi_2\rangle) + \cdots + a_k(U|\psi_k\rangle). \tag{B.17}$$

To fulfil these properties, we require that U *preserve the inner product*:

$$\langle \psi_0 | U^\dagger U | \psi \rangle = \langle \psi_0 | \psi \rangle$$

That is, we require that any such transformation be *unitary*:

$$UU^\dagger = I \Rightarrow U^\dagger = U^{-1} \tag{B.18}$$

Unitarity is a sufficient condition to describe any valid quantum operation: any quantum transformation can be described by a unitary transformation, and any unitary transformation corresponds to a physically implementable quantum transformation.

Then, if U_1 is a unitary transformation that acts on the space $\mathcal{H}_1$ and U_2 acts on $\mathcal{H}_2$, the product of the two unitary transformations is also unitary. The tensor product $U_1 \otimes U_2$ acts on the space $\mathcal{H}_1 \otimes \mathcal{H}_2$. So, then, supposing a system of two separable qubits, $|\psi_1\rangle$ and $|\psi_2\rangle$ where we wish to act on $|\psi_1\rangle$ with operator U_1 and on $|\psi_2\rangle$ with U_2, we perform it as

$$(U_1 \otimes U_2)(|\psi_1\rangle \otimes |\psi_2\rangle) = (U_1 |\psi_1\rangle) \otimes (U_2 |\psi_2\rangle) \tag{B.19}$$

B.6 DIRAC NOTATION

In keeping with standard practice, we employ *Dirac notation* throughout this thesis. Vectors are denoted by *kets* of the form $|a\rangle$. For example, the standard basis is represented by,

$$\begin{aligned} |x\rangle = |0\rangle &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ |y\rangle = |1\rangle &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{aligned} \tag{B.20}$$

We saw in Table B.1 that for every such ket, $|\psi\rangle$, there exists a *dual vector*: its complex conjugate transpose, called the *bra* of such a vector, denoted $\langle \psi |$. That is,

$$\begin{aligned} \langle \psi |^\dagger &= |\psi\rangle \\ |\psi\rangle^\dagger &= \langle \psi | \end{aligned} \tag{B.21}$$

$$|\psi\rangle = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_n \end{pmatrix} \Rightarrow \langle \psi | = \begin{pmatrix} \psi_1^* & \psi_2^* & \dots & \psi_n^* \end{pmatrix} \tag{B.22}$$

Then if we have two vectors $|\psi\rangle$ and $|\phi\rangle$, their *inner product* is given as $\langle\psi|\phi\rangle = \langle\phi|\psi\rangle$.

$$\begin{aligned} |\psi\rangle = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \vdots \\ \psi_n \end{pmatrix} \quad ; \quad |\phi\rangle = \begin{pmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \phi_n \end{pmatrix} \\ \Rightarrow \langle\phi| = \begin{pmatrix} \phi_1^* & \phi_2^* & \phi_3^* & \dots & \phi_n^* \end{pmatrix} \\ \Rightarrow \langle\phi|\,|\psi\rangle = \begin{pmatrix} \phi_1^* & \phi_2^* & \phi_3^* & \dots & \phi_n^* \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \vdots \\ \psi_n \end{pmatrix} \\ \Rightarrow \langle\phi|\,|\psi\rangle = \phi_1^*\psi_1 + \phi_2^*\psi_2 + \phi_3^*\psi_3 + \cdots + \phi_n^*\psi_n \end{aligned} \tag{B.23}$$

Example B.6.1.

$$\begin{aligned} |\psi\rangle = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad ; \quad |\phi\rangle = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \\ \Rightarrow \langle\phi|\,|\psi\rangle = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \\ = (4)(1) + (5)(2) + (6)(3) = 32 \end{aligned} \tag{B.24}$$

Similarly, their *outer product* is given as $|\phi\rangle\langle\psi|$. Multiplying a column vector by a row vector thus gives a matrix. Matrices generated by a outer products then define operators:

Example B.6.2.

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} \begin{pmatrix} 3 & 4 \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ 6 & 8 \end{pmatrix} \tag{B.25}$$

Then we can say, for $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

$$|0\rangle\langle 0| = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \tag{B.26a}$$

$$|0\rangle\langle 1| = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \tag{B.26b}$$

$$|1\rangle\langle 0| = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \tag{B.26c}$$

$$|1\rangle\langle 1| = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{B.26d}$$

And so any 2-dimensional linear transformation in the standard basis $|0\rangle, |1\rangle$ can be given as a sum

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = a|0\rangle\langle 0| + b|0\rangle\langle 1| + c|1\rangle\langle 0| + d|1\rangle\langle 1| \tag{B.27}$$

This is a common method of representing operators as outer products of vectors. A transformation that *exchanges* a particle between two states, say $|0\rangle \leftrightarrow |1\rangle$ is given by the operation

$$\hat{Q} : \begin{cases} |0\rangle \to |1\rangle \\ |1\rangle \to |0\rangle \end{cases}$$

Which is equivalent to the outer product representation

$$\hat{Q} = |0\rangle\langle 1| + |1\rangle\langle 0|$$

For clarity, here we will prove this operation

Example B.6.3.

$$\begin{aligned} \hat{Q} &= |0\rangle\langle 1| + |1\rangle\langle 0| \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \end{aligned}$$

So then, acting on $|0\rangle$ *and* $|1\rangle$ *gives*

$$\hat{Q}|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$$

$$\hat{Q}\left|1\right\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \left|0\right\rangle$$

To demonstrate how Dirac notation simplifies this:

$$\begin{aligned} \hat{Q}\left|0\right\rangle &= \left(\left|0\right\rangle\left\langle 1\right| + \left|1\right\rangle\left\langle 0\right|\right)\left|0\right\rangle \\ &= \left|0\right\rangle\left\langle 1\right|\left|0\right\rangle + \left|1\right\rangle\left\langle 0|0\right\rangle \\ &= \left|0\right\rangle\left\langle 1|0\right\rangle + \left|1\right\rangle\left\langle 0|0\right\rangle \end{aligned}$$

Then, since $\left|0\right\rangle$ and $\left|1\right\rangle$ are orthogonal basis, their inner product is 0 and the inner product of a vector with itself is 1, ($\left\langle 1|1\right\rangle = \left\langle 0|0\right\rangle = 1, \left\langle 0|1\right\rangle = \left\langle 1|0\right\rangle = 0$). So,

$$\begin{aligned} \hat{Q}\left|0\right\rangle &= \left|0\right\rangle(0) + \left|1\right\rangle(1) \\ \Rightarrow \hat{Q}\left|0\right\rangle &= \left|1\right\rangle \end{aligned} \tag{B.28}$$

And similarly for $\hat{Q}\left|1\right\rangle$. This simple example then shows why Dirac notation can significantly simplify calculations across quantum mechanics, compared to standard matrix and vector notation. To see this more clearly, we will examine a simple 2-qubit state under such operations. The method generalises to operating on two or more qubits generically: we can define any operator which acts on two qubits as a sum of outer products of the basis vectors $\left|00\right\rangle, \left|01\right\rangle, \left|10\right\rangle$ and $\left|11\right\rangle$. We can similarly define any operator which acts on an n qubit state as a linear combination of the 2^n basis states generated by the n qubits.

Example B.6.4. *To define a transformation that will exchange basis vectors $\left|00\right\rangle$ and $\left|11\right\rangle$, while leaving $\left|01\right\rangle$ and $\left|10\right\rangle$ unchanged (ie exchanging $\left|01\right\rangle \leftrightarrow \left|01\right\rangle, \left|10\right\rangle \leftrightarrow \left|10\right\rangle$) we define an operator*

$$\hat{Q} = \left|00\right\rangle\left\langle 11\right| + \left|11\right\rangle\left\langle 00\right| + \left|10\right\rangle\left\langle 10\right| + \left|01\right\rangle\left\langle 01\right| \tag{B.29}$$

Then, using matrix calculations this would require separately calculating the four outer products in the above sum and adding them to find a 4×4 matrix to represent $\hat{Q}$, which then acts on a state $\left|\psi\right\rangle$. Instead, consider first that $\left|\psi\right\rangle = \left|00\right\rangle$, ie one of the basis vectors our transformation is to change:

$$\hat{Q}\left|00\right\rangle = \left(\left|00\right\rangle\left\langle 11\right| + \left|11\right\rangle\left\langle 00\right| + \left|10\right\rangle\left\langle 10\right| + \left|01\right\rangle\left\langle 01\right|\right)\ \left|00\right\rangle \tag{B.30}$$

And as before, only the inner products of a vector with itself remains:

$$\begin{aligned} &= \left|00\right\rangle\left\langle 11\left|00\right.\right\rangle + \left|11\right\rangle\left\langle 00\left|00\right.\right\rangle + \left|10\right\rangle\left\langle 10\left|00\right.\right\rangle + \left|01\right\rangle\left\langle 01\left|00\right.\right\rangle \\ &= \left|00\right\rangle(0) + \left|11\right\rangle(1) + \left|10\right\rangle(0) + \left|01\right\rangle(0) \\ \Rightarrow &\hat{Q}\left|00\right\rangle = \left|11\right\rangle \end{aligned} \tag{B.31}$$

i.e the transformation has performed $\hat{Q} : |00\rangle \rightarrow |11\rangle$ *as expected. Then, if we apply the same transformation to a state which does not depend on one of the target states, eg,*

$$\begin{aligned} |\psi\rangle =& a\,|10\rangle + b\,|01\rangle \\ \hat{Q}\,|\psi\rangle =& \Big(|00\rangle\langle 11| + |11\rangle\langle 00| + |10\rangle\langle 10| + |01\rangle\langle 01| \Big)\Big(a\,|10\rangle + b\,|01\rangle \Big) \\ =& a\Big(|00\rangle\langle 11|10\rangle + |11\rangle\langle 00|10\rangle + |10\rangle\langle 10|10\rangle + |01\rangle\langle 01|10\rangle \Big) \\ &+ b\Big(|00\rangle\langle 11|01\rangle + |11\rangle\langle 00|01\rangle + |10\rangle\langle 10|01\rangle + |01\rangle\langle 01|01\rangle \Big) \end{aligned} \tag{B.32}$$

And since the inner product is a scalar, we can factor terms such as $\langle 11|10\rangle$ *to the beginning of expressions, eg* $|00\rangle\langle 11|\,|10\rangle = \langle 11\,|10\rangle\,|00\rangle$*, and we also know*

$$\begin{aligned} \langle 11|10\rangle = \langle 00|10\rangle = \langle 01|10\rangle = \langle 11|01\rangle = \langle 00|01\rangle = \langle 10|01\rangle = 0 \\ \langle 10|10\rangle = \langle 01|01\rangle = 1 \end{aligned} \tag{B.33}$$

We can express the above as

$$\begin{aligned} \hat{Q}\,|\psi\rangle =& a\Big((0)\,|00\rangle + (0)\,|11\rangle + (1)\,|10\rangle + (0)\,|01\rangle \Big) \\ &+ b\Big((0)\,|00\rangle + (0)\,|11\rangle + (0)\,|10\rangle + (1)|01\rangle \Big) \\ =& a\,|10\rangle\,| + b|01\rangle \\ =& |\psi\rangle \end{aligned} \tag{B.34}$$

Then it is clear that, when $|\psi\rangle$ *is a superposition of states unaffected by transformation* $\hat{Q}$*, then* $\hat{Q}\,|\psi\rangle = |\psi\rangle$*.*

This method generalises to systems with greater numbers of particles (qubits). If we briefly consider a 3 qubit system - and initialise all qubits in the standard basis state $|0\rangle$ - then the system is represented by $|000\rangle = |0\rangle \otimes |0\rangle \otimes |0\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. This quantity is an 8-row vector. To calculate the outer product $\langle 000|000\rangle$, we would be multiplying an 8-column bra $\langle 000|$ by an 8-row ket $|000\rangle$. Clearly then we will be working with 8×8 matrices, which will become quite difficult to maintain effectively and efficiently quite fast. As we move to systems of larger size, standard matrix multiplication becomes impractical for hand-written analysis, although of course remains tractable computationally up to $n \sim 10$ qubits. It is obvious that Dirac's bra/ket notation is a helpful, pathematically precise tool for QM.

C

EXAMPLE EXPLORATION STRATEGY RUN

Here we provide a complete example of how to run the Quantum Model Learning Agent (QMLA) framework, including how to implement a custom exploration strategy (ES), and generate/interpret analysis. Note: these examples are included in the QMLA documentation in a format that may be easier to follow – where possible, we recommend readers follow the Tutorial section of [30].

First, *fork* the QMLA codebase from [31] to a Github user account (referred to as username in Listing C.6). Now, we must download the code base and ensure it runs properly; these instructions are implemented via the command line[1].

The steps of preparing the codebase are

1. install redis;
2. create a virtual Python environment for installing QMLA dependencies without damaging other parts of the user's environment;
3. download the QMLA codebase from the forked Github repository;
4. install packages upon which QMLA depends.

```
# Install redis (database broker)
sudo apt update
sudo apt install redis-server

# make directory for QMLA
cd
mkdir qmla_test
cd qmla_test

# make Python virtual environment for QMLA
# note: change Python3.6 to desired version
sudo apt-get install python3.6-venv
python3.6 -m venv qmla-env
source qmla-env/bin/activate
```

1 Note: these instructions are tested for Linux and presumed to work on Mac, but untested on Windows. It is likely some of the underlying software (redis servers) can not be installed on Windows, so running on *Windows Subsystem for Linux* is advised.

```
# Download QMLA
git clone --depth 1 https://github.com/username/QMLA.git #
    REPLACE username

# Install dependencies
cd QMLA
pip install -r requirements.txt
```

Listing C.1: QMLA codebase setup language

Note there may be a problem with some packages in the requirements.txt arising from the attempt to install them all through a single call to pip install. Ensure these are all installed before proceeding.

When all of the requirements are installed, test that the framework runs. QMLA uses redis databases to store intermittent data: we must manually initialise the database. Run the following (note: here we list redis-4.0.8, but this must be corrected to reflect the version installed on the user's machine in the above setup section):

```
~/redis-4.0.8/src/redis-server
```

Listing C.2: Launch redis database

which should give something like Fig. C.1.

Figure C.1: Terminal running redis-server.

In a text editor, open qmla_test/QMLA/launch/local_launch.sh; here we will ensure that we are running the QHL algorithm, with 5 experiments and 20 particles, on the ES named TestInstall. Ensure the first few lines of local_launch.sh read:

```
#!/bin/bash

##### ---------------------------------------------- #####
# QMLA run configuration
##### ---------------------------------------------- #####
num_instances=2 # number of instances in run
run_qhl=0 # perform QHL on known (true) model
run_qhl_multi_model=0 # perform QHL for defined list of models
experiments=2 # number of experiments
particles=10 # number of particles
plot_level=5


##### ---------------------------------------------- #####
# Choose an exploration strategy
# This will determine how QMLA proceeds.
##### ---------------------------------------------- #####
exploration_strategy="TestInstall"
```

Listing C.3: local_launch script

Ensure the terminal running redis is kept active, and open a separate terminal window. We must activate the Python virtual environment configured for QMLA, which we set up in Listing C.6. Then, we navigate to the QMLA directory, and launch:

```
# activate the QMLA Python virtual environment
source qmla_test/qmla-env/bin/activate

# move to the QMLA directory
cd qmla_test/QMLA
# Run QMLA
cd launch
./local_launch.sh
```

Listing C.4: Launch QMLA

There may be numerous warnings, but they should not affect whether QMLA has succeeded; QMLA will raise any significant error. Assuming the run has completed successfully, QMLA stores the run's results in a subdirectory named by the date and time it was started. For example, if the run was initialised on January 1st at 01:23, navigate to the corresponding directory by

```
cd results/Jan_01/01_23
```

Listing C.5: QMLA results directory

For now it is sufficient to notice that the code has run successfully: it should have generated (in results/Jan_01/01_23) files like storage_001.p and results_001.p.

C.1 CUSTOM EXPLORATION STRATEGY

Next, we design a basic ES, for the purpose of demonstrating how to run the algorithm. ESs are placed in the directory qmla/exploration_strategies. To make a new one, navigate to the exploration strategies directory, make a new subdirectory, and copy the template file.

```
cd ~/qmla_test/QMLA/exploration_strategies/
mkdir custom_es

# Copy template file into example
cp template.py custom_es/example.py
cd custom_es
```

Listing C.6: QMLA codebase setup

Ensure QMLA will know where to find the ES by importing everything from the custom ES directory into to the main exploration_strategy module. Then, in the custom_es directory, make a file called __init__.py which imports the new ES from the example.py file. To add any further ESs inside the directory custom_es, include them in the custom __init__.py, and they will automatically be available to QMLA.

```
# inside qmla/exploration_strategies/custom_es
#   __init__.py
from qmla.exploration_strategies.custom_es.example import *

# inside qmla/exploration_strategies , add to the existing
# __init__.py
from qmla.exploration_strategies.custom_es import *
```

Listing C.7: Providing custom exploration strategy to QMLA

Now, change the structure (and name) of the ES inside custom_es/example.py. Say we wish to target the true model

$$\begin{aligned} \vec{\alpha} &= (\alpha_{1,2} \quad \alpha_{2,3} \quad \alpha_{3,4}) \\ \vec{T} &= \begin{pmatrix} \hat{\sigma}_z^1 \otimes \hat{\sigma}_z^2 \\ \hat{\sigma}_z^2 \otimes \hat{\sigma}_z^3 \\ \hat{\sigma}_z^3 \otimes \hat{\sigma}_z^4 \end{pmatrix} \\ \implies \hat{H}_0 &= \hat{\sigma}_z^{(1,2)} \hat{\sigma}_z^{(2,3)} \hat{\sigma}_z^{(3,4)} \end{aligned} \tag{C.1}$$

QMLA interprets models as strings, where terms are separated by $+$, and parameters are implicit. So the target model in Eq. (C.1) will be given by

pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4+pauliSet_3J4_zJz_d4.

Adapting the template ES slightly, we can define a model generation strategy with a small number of hard coded candidate models introduced at the first branch of the exploration tree. We will also set the parameters of the terms which are present in $\hat{H}_0$, as well as the range in which to search parameters. Keeping the imports at the top of the example.py, rewrite the ES as:

```
class ExampleBasic(
    exploration_strategy.ExplorationStrategy
):

    def __init__(
        self,
        exploration_rules,
        true_model=None,
        **kwargs
    ):
        self.true_model = 'pauliSet_1J2_zJz_d4+
            pauliSet_2J3_zJz_d4+pauliSet_3J4_zJz_d4'
        super().__init__(
            exploration_rules=exploration_rules,
            true_model=self.true_model,
            **kwargs
        )

        self.initial_models = None
        self.true_model_terms_params = {
            'pauliSet_1J2_zJz_d4' : 2.5,
```

```
            'pauliSet_2J3_zJz_d4' : 7.5,
            'pauliSet_3J4_zJz_d4' : 3.5,
        }
        self.tree_completed_initially = True
        self.min_param = 0
        self.max_param = 10

    def generate_models(self, **kwargs):

        self.log_print(["Generating models; spawn step {}".format
            (self.spawn_step)])
        if self.spawn_step == 0:
            # chains up to 4 sites
            new_models = [
                'pauliSet_1J2_zJz_d4',
                'pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4',
                'pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4+
                    pauliSet_3J4_zJz_d4',
            ]
            self.spawn_stage.append('Complete')

        return new_models
```

Listing C.8: ExampleBasic exploration strategy.

To run[2] the example ES for a meaningful test, return to the local_launch of Listing C.3, but change some of the settings:

```
particles=2000
experiments=500
run_qhl=1
exploration_strategy=ExampleBasic
```

Listing C.9: local_launch configuration for QHL.

Run locally again as in Listing C.4; then move to the results directory as in Listing C.5.

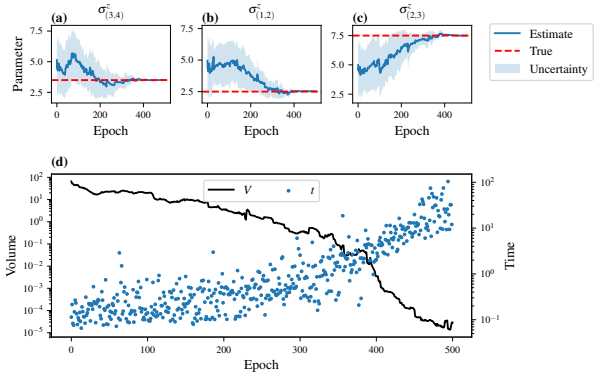
2 Note this will take up to 15 minutes to run. This can be reduced by lowering the values of particles, experiments, which is sufficient for testing but note that the outcomes will be less effective than those presented in the figures of this section.

C.2 ANALYSIS

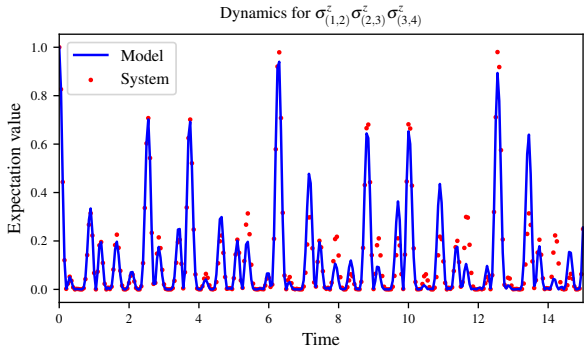
QMLA stores results and generates plots over the entire range of the algorithm[3], i.e. the run, instance and models. The depth of analysis performed automatically is set by the user control plot_level in local_launch.sh; for plot_level=1, only the most crucial figures are generated, while plot_level=6 generates plots for every individual model considered. For model searches across large model spaces and/or considering many candidates, excessive plotting can cause considerable slow-down, so users should be careful to generate plots only to the degree they will be useful. Next we show some examples of the available plots.

C.2.1 *Model analysis*

We have just run quantum Hamiltonian learning (QHL) for the model in Eq. (C.1) for a single instance, using a reasonable number of particles and experiments, so we expect to have trained the model well. Instance-level results are stored (e.g. for the instance with qmla_id=1) in Jan_01/01_23/instances/qmla_1. Individual models' insights can be found in model_training, e.g. the model's learning_summary Fig. C.2a, and dynamics in Fig. C.2b.



(a)



(b)

Figure C.2: Model analysis plots, stored in (for example) Jan_01/01_23/instances/qmla_1/model_training. **a** learning_summary_1. Displays the outcome of QHL for the given model: Subfigures (a)-(c) show the estimates of the parameters; (d) shows the total parameterisation volume against experiments trained upon, along with the evolution times used for those experiments. **(b)** dynamics_1 The model's attempt at reproducing dynamics from $\hat{H}_0$.

3 Recall that a single implementation of QMLA is called an instance, while a series of instances – which share the same target model – is called the run.

C.2.2 *Instance analysis*

Now we can run the full QMLA algorithm, i.e. train several models and determine the most suitable. QMLA will call the generate_models method of the ExampleBasic ES, set in Listing C.8, which tells QMLA to construct three models on the first branch, then terminate the search. Here we need to train and compare all models so it takes considerably longer to run: for the purpose of testing, we reduce the resources so the entire algorithm runs in about 15 minutes. Some applications will require significantly more resources to learn effectively. In realistic cases, these processes are run in parallel, as we will cover in Appendix C.3.

Reconfigure a subset of the settings in the local_launch.sh script (Listing C.3) and run it again:

```
experiments=250
particles=1000
run_qhl=0
exploration_strategy=ExampleBasic
```

Listing C.10: local_launch configuration for QMLA.

In the corresponding results directory, navigate to instances/qmla_1, where instance level analysis are available.

```
cd results/Jan_01/01_23/instances/qmla_1
```

Listing C.11: Navigating to instance results.

Figures of interest here show the composition of the models (Fig. C.3a), as well as the Bayes factors between candidates (Fig. C.3b). Individual model comparisons – i.e. Bayes factor (BF) – are shown in Fig. C.3c, with the dynamics of all candidates shown in Fig. C.4c. The probes used during the training of all candidates are also plotted (Fig. C.3e).

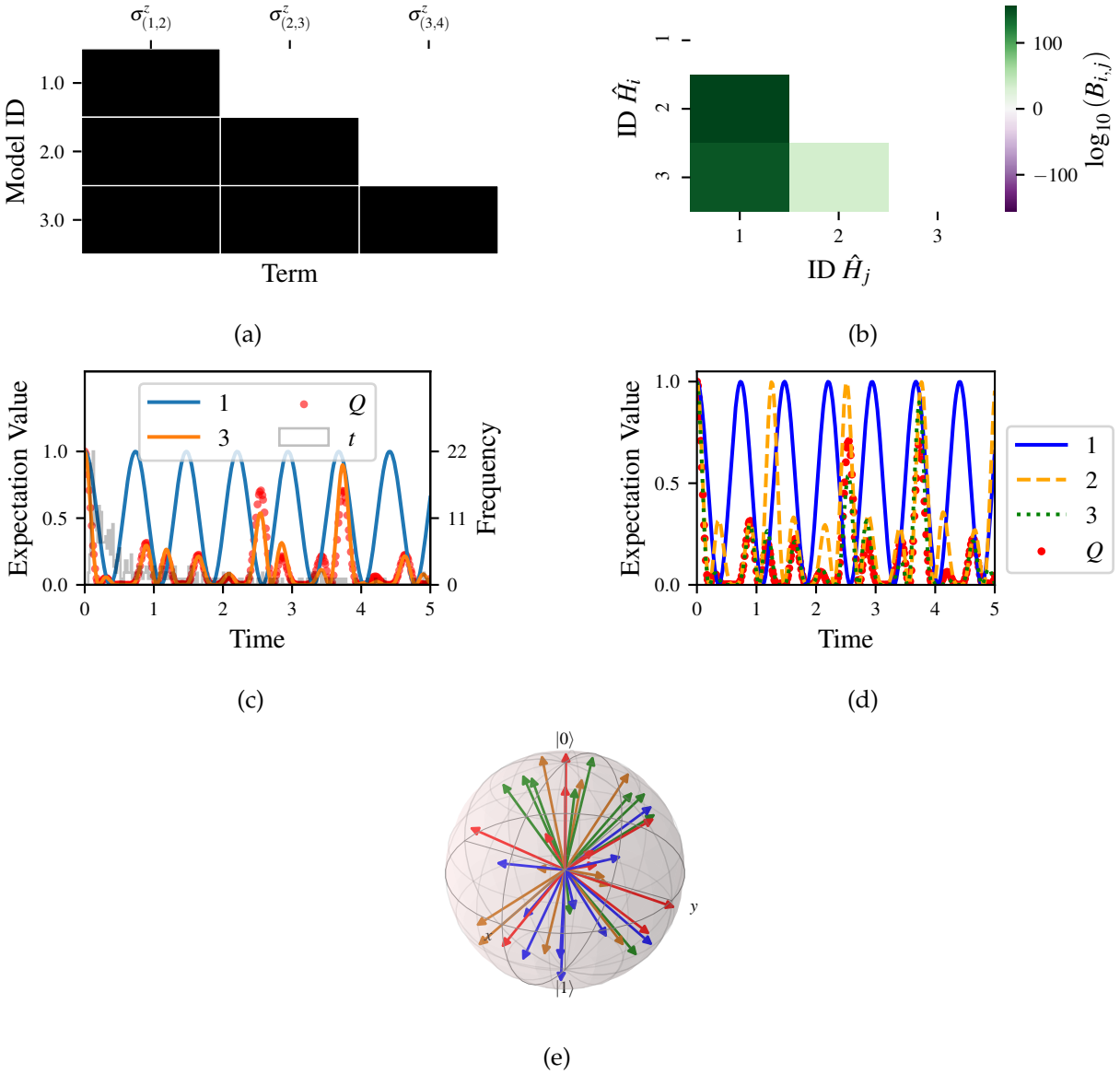


Figure C.3: QMLA plots; found within instance directory e.g. Jan_01/01_23/instances/qmla_1, and its subdirectories. **(a)** composition_of_models: constituent terms of all considered models, indexed by their model IDs. Here model 3 is $\hat{H}_0$. **(b)** bayes_factors: Bayes factor (BF) comparisons between all models. Bayes factors (BFs) are read as $B_{i,j}$ where i is the model with lower ID, e.g. $B_{1,2}$ rather than $B_{2,1}$. Thus $B_{ij} > 0$ (< 0) indicates $\hat{H}_i$ ($\hat{H}_j$), i.e. the model on the y-axis (x-axis) is the stronger model. **(c)** comparisons/BF_1_3: direct comparison between models with IDs 1 and 3, showing their reproduction of the system dynamics (red dots, Q), as well as the times (experiments) against which the BF was calculated. **(d)** branches/dynamics_branch_1: dynamics of all models considered on the branch compared with system dynamics (red dots, Q). **(e)** probes_bloch_sphere: probes used for training models in this instance (only showing 1-qubit versions).

C.2.3 *Run analysis*

Considering a number of instances together is a *run*. In general, this is the level of analysis of most interest: an individual instance is liable to errors due to the probabilistic nature of the model training and generation subroutines. On average, however, we expect those elements to perform well, so across a significant number of instances, we expect the average outcomes to be meaningful.

Each results directory has an analyse.sh script to generate plots at the run level.

```
cd results/Jan_01/01_23
./analyse.sh
```

Listing C.12: Analysing QMLA run.

Run level analysis are held in the main results directory and several sub-directories created by the analyse script. Here, we recommend running a number of instances with very few resources so that the test finishes quickly[4]. The results will therefore be meaningless, but allow fo elucidation of the resultant plots. First, reconfigure some settings of Listing C.3 and launch again.

```
num_instances=10
experiments=20
particles=100
run_qhl=0
exploration_strategy=ExampleBasic
```

Listing C.13: local_launch configuration for QMLA run.

Some of the generated analysis are shown in Figs. C.4 to C.5. The number of instances for which each model was deemed champion, i.e. their *win rates* are given in Fig. C.4a. The *top models*, i.e. those with highest win rates, analysed further: the average parameter estimation progression for $\hat{H}_0$ – including only the instances where $\hat{H}_0$ was deemed champion – are shown in Fig. C.4b. Irrespecitve of the champion models, the rate with which each term is found in the champion model ($\hat{t} \in \hat{H}'$) indicates the likelihood that the term is really present; these rates – along with the parameter values learned – are shown in Fig. C.4c. The champion model from each instance can attempt to reproduce system dynamics: we group together these reproductions for each model in Fig. C.5.

4 This run will take about ten minutes

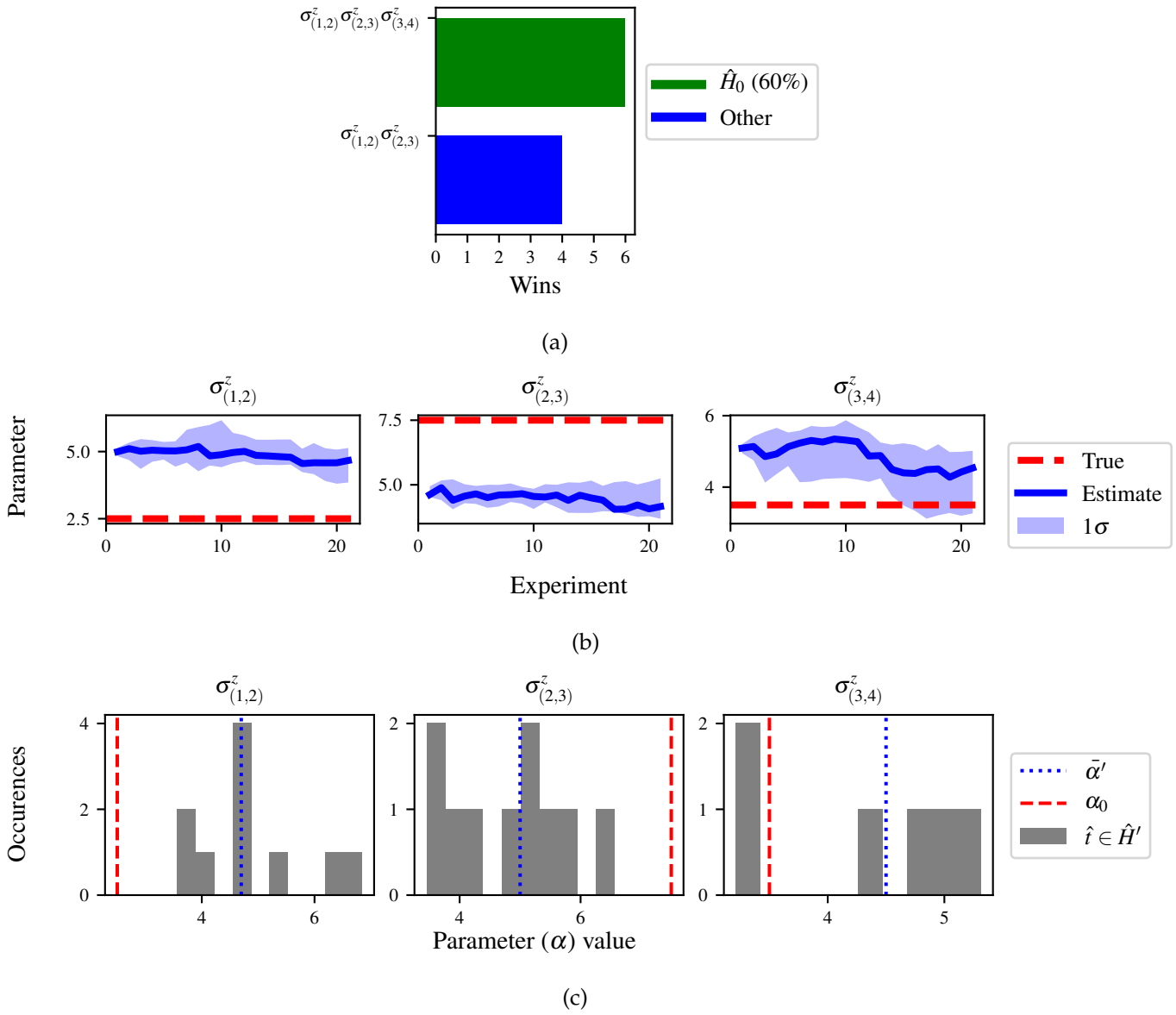


Figure C.4: QMLA run plots; found within run directory e.g. Jan_01/01_23/. **(a)** performance/model_wins: number of instance wins achieved by each model. **(b)** champion_models/params_params_pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4+pauliSet_3J4_zJz_d4: parameter estimation progression for the true model , only for the instances where it was deemed champion. **(c)** champion_models/terms_and_params: histogram of parameter values found for each term which appears in any champion model, with the true parameter (α_0) in red and the median learned parameter ($\bar{\alpha}'$) in blue.

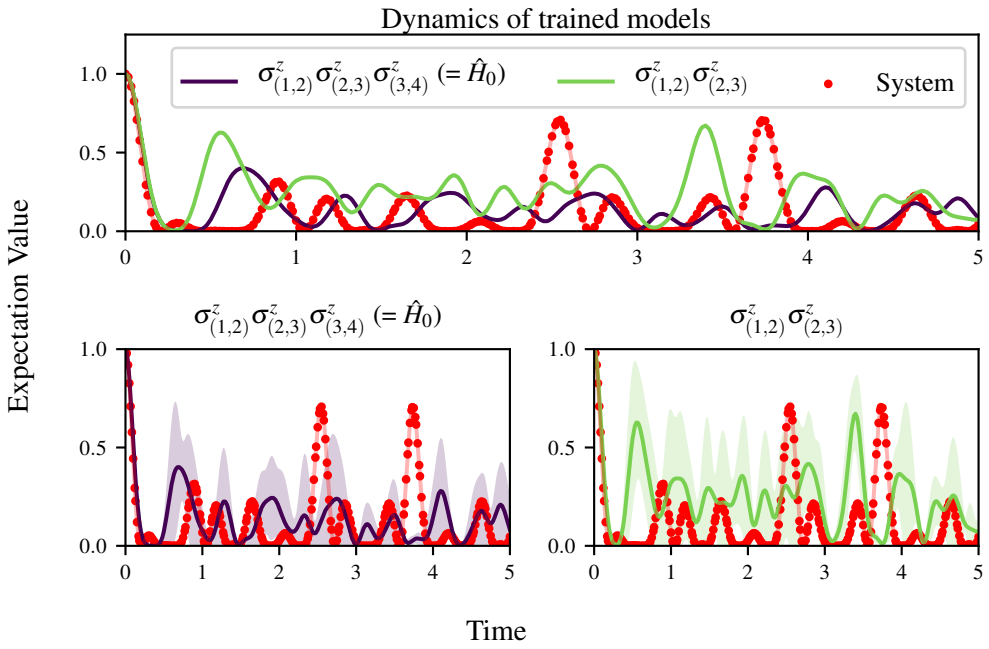


Figure C.5: Run plot performace/dynamics: median dynamics of the champion models. The models which won most instances are shown together in the top panel, and individually in the lower panels. The median dynamics from the models' learnings in its winning instances are shown, with the shaded region indicating the 66% confidence region.

C.3 PARALLEL IMPLEMENTATION

We provide utility to run QMLA on parallel processes. Individual models' training can run in parallel, as well as the calculation of BF between models. The provided script is designed for portable batch system (PBS) job scheduler running on a compute cluster. It will require a few adjustments to match the system being used. Overall, though, it has mostly a similar structure as the local_launch.sh script used above.

QMLA must be downloaded on the compute cluster as in Listing C.6; this can be a new fork of the repository, though it is sensible to test installation locally as described in this chapter so far, then *push* that version, including the new ES, to Github, and cloning the latest version. It is again advisable to create a Python virtual environment in order to isolate QMLA and its dependencies[5]. Open the parallel launch script, QMLA/launch/parallel_launch.sh, and prepare the first few lines as

```
#!/bin/bash
```

5 Indeed it is sensible to do this for any Python development project.

```
##### ---------------------------------------------- #####
# QMLA run configuration
##### ---------------------------------------------- #####
num_instances=10 # number of \glspl{instance} in run
run_qhl=0 # perform QHL on known (true) model
run_qhl_multi_model=0 # perform QHL for defined list of models
experiments=250
particles=1000
plot_level=5


##### ---------------------------------------------- #####
# Choose an exploration strategy
# This will determine how QMLA proceeds.
##### ---------------------------------------------- #####
exploration_strategy="ExampleBasic"
```

Listing C.14: parallel_launch script

When submitting jobs to schedulers like PBS, we must specify the time required, so that it can determine a fair distribution of resources among users. We must therefore *estimate* the time it will take for an instance to complete: clearly this is strongly dependent on the numbers of experiments (N_e) and particles (N_p), and the number of models which must be trained. QMLA attempts to determine a reasonable time to request based on the max_num_models_by_shape attribute of the ES, by calling QMLA/scripts/time_required_calculation.py. In practice, this can be difficult to set perfectly, so the timing_insurance_factor attribute of the ES can be used to correct for heavily over- or under-estimated time requests. Instances are run in parallel, and each instance trains/compares models in parallel. The number of processes to request, N_c for each instance is set as num_processes_to_parallelise_over in the ES. Then, if there are N_r instances in the run, we will be requesting the job scheduler to admit N_r distinct jobs, each requiring N_c processes, for the time specified.

The parallel_launch script works together with launch/run_single_qmla_instance.sh, though note a number of steps in the latter are configured to the cluster and may need to be adapted. In particular, the first command is used to load the redis utility, and later lines are used to initialise a redis server. These commands will probably not work with most machines, so must be configured to achieve those steps.

```
module load tools/redis-4.0.8

...
```

```
SERVER_HOST=$(head -1 "$PBS_NODEFILE")
let REDIS_PORT="6300 + $QMLA_ID"

cd $LIBRARY_DIR
redis-server RedisDatabaseConfig.conf --protected-mode no --port
    $REDIS_PORT &
redis-cli -p $REDIS_PORT flushall
```

Listing C.15: run_single_qmla_instance script

When the modifications are finished, QMLA can be launched in parallel similarly to the local version:

```
source qmla_test/qmla-env/bin/activate

cd qmla_test/QMLA/launch
./parallel_launch.sh
```

Listing C.16: run_single_qmla_instance script

Jobs are likely to queue for some time, depending on the demands on the job scheduler. When all jobs have finished, results are stored as in the local case, in QMLA/launch/results/-Jan_01/01_23, where analyse.sh can be used to generate a series of automatic analyses.

C.4 CUSTOMISING EXPLORATION STRATEGYS

User interaction with the QMLA codebase should be achieveable primarily through the exploration strategy (ES) framework. Throughout the algorithm(s) available, QMLA calls upon the ES before determining how to proceed. The usual mechanism through which the actions of QMLA are directed, is to set attributes of the ES class: the complete set of influential attributes are available at [30].

QMLA directly uses several methods of the ES class, all of which can be overwritten in the course of customising an ES. Most such methods need not be replaced, however, with the exception of generate_models, which is the most important aspect of any ES: it determines which models are built and tested by QMLA. This method allows the user to impose any logic desired in constructing models; it is called after the completion of every branch of the exploration tree on the ES.

C.4.1 *Greedy search*

A first non-trivial ES is to build models greedily from a set of *primitive* terms, $\mathcal{T} = \{\hat{t}\}$. New models are constructed by combining the previous branch champion with each of the remaining, unused terms. The process is repeated until no terms remain.

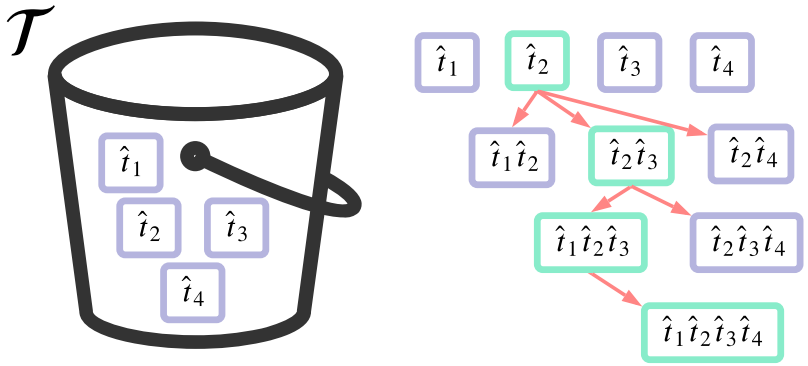


Figure C.6: Greedy search mechanism. **Left**, a set of primitive terms, $\mathcal{T}$, are defined in advance. **Right**, models are constructed from $\mathcal{T}$. On the first branch, the primitve terms alone constitute models. Thereafter, the strongest model (marked in green) from the previous branch is combined with all the unused terms.

We can compose an ES using these rules, say for

$$\mathcal{T} = \left\{\hat{\sigma}_x^1,\ \hat{\sigma}_y^1,\ \hat{\sigma}_x^1 \otimes \hat{\sigma}_x^2,\ \hat{\sigma}_y^1 \otimes \hat{\sigma}_y^2\right\}$$

as follows. Note the termination criteria must work in conjunction with the model generation routine. Users can overwrite the method check_tree_completed for custom logic, although a straightforward mechanism is to use the spawn_stage attribute of the ES class: when the final element of this list is Complete, QMLA will terminate the search by default. Also note that the default termination test checks whether the number of branches (spawn_step) exceeds the limit max_spawn_depth, which must be set artifically high to avoid ceasing the search too early, if relying solely on spawn_stage. Here we demonstrate how to impose custom logic to terminate the seach also.

```
class ExampleGreedySearch(
    exploration_strategy.ExplorationStrategy
):
    r"""
    From a fixed set of terms, construct models iteratively,
```

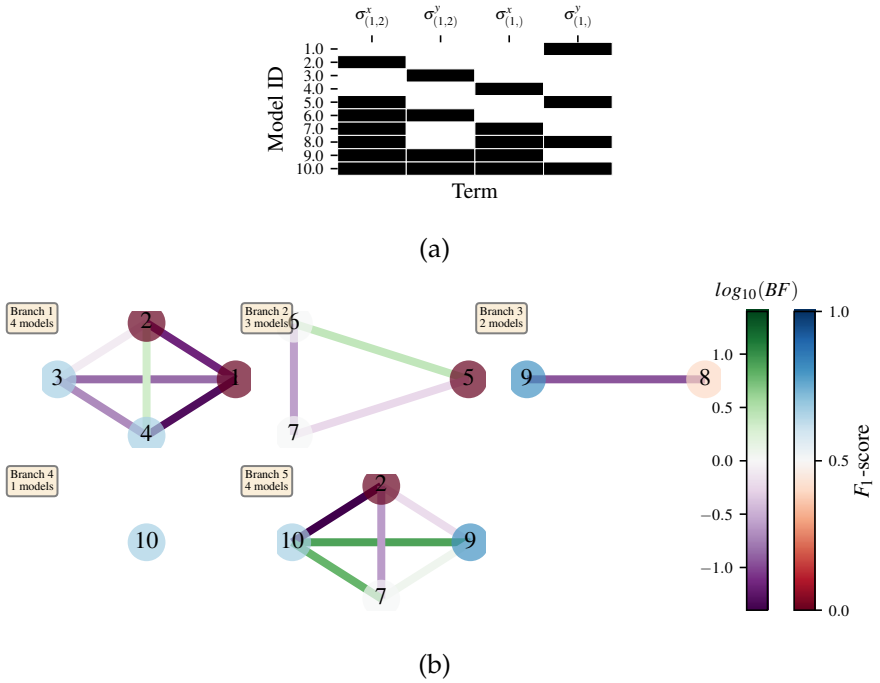
```
    greedily adding all unused terms to separate models at each
       call to the generate_models.

    """

    def __init__(
        self,
        exploration_rules,
        **kwargs
    ):

        super().__init__(
            exploration_rules=exploration_rules,
            **kwargs
        )
        self.true_model = 'pauliSet_1_x_d3+pauliSet_1J2_yJy_d3+
           pauliSet_1J2J3_zJzJz_d3'
        self.initial_models = None
        self.available_terms = [
            'pauliSet_1_x_d3', 'pauliSet_1_y_d3',
            'pauliSet_1J2_xJx_d3', 'pauliSet_1J2_yJy_d3'
        ]
        self.branch_champions = []
        self.prune_completed_initially = True
        self.check_champion_reducibility = False

    def generate_models(
        self,
        model_list,
        **kwargs
    ):
        self.log_print([
            "Generating models in tiered greedy search at spawn
               step {}.".format(
                 self.spawn_step,
            )
        ])
        try:
            previous_branch_champ = model_list[0]
            self.branch_champions.append(previous_branch_champ)
```

```
        except:
            previous_branch_champ = ""

        if self.spawn_step == 0 :
            new_models = self.available_terms
        else:
            new_models = greedy_add(
                current_model = previous_branch_champ,
                terms = self.available_terms
            )

        if len(new_models) == 0:
            # Greedy search has exhausted the available models;
            # send back the list of branch champions and
               terminate search.
            new_models = self.branch_champions
            self.spawn_stage.append('Complete')

        return new_models

def greedy_add(
    current_model,
    terms,
):
    r"""
    Combines given model with all terms from a set.

    Determines which terms are not yet present in the model,
    and adds them each separately to the current model.

    :param str current_model: base model
    :param list terms: list of strings of terms which are to be
       added greedily.
    """


    try:
        present_terms = current_model.split('+')
    except:
        present_terms = []
    nonpresent_terms = list(set(terms) - set(present_terms))
```

```
    term_sets = [
        present_terms+[t] for t in nonpresent_terms
    ]

    new_models = ["+".join(term_set) for term_set in term_sets]

    return new_models
```

Listing C.17: ExampleGreedySearch exploration stategy

This run can be implemented locally or in parallel as described above[6], and analysed as in Listing C.12, generating figures in accordance with the plot_level set by the user in the launch script. Outputs can again be found in the instances subdirectory, including a map of the models generated, as well as the branches they reside on, and the BFs between candidates, Fig. C.7.



(a)

(b)

Figure C.7: Greedy exploration strategy. **(a)**, composition_of_models. **b**, graphs_of_branches_ExampleGreedySearch: shows which models reside on each branches of the exploration tree. Models are coloured by their F_1-score, and edges represent the BF between models. The first four branches are equivalent to those in Fig. C.6 , while the final branch considers the set of branch champions, in order to determine the overall champion.

6 We advise reducing plot_level to 3 to avoid excessive/slow figure generation.

C.4.2 *Tiered greedy search*

We provide one final example of a non-trivial ES: tiered greedy search. Similar to the idea of Appendix C.4.1, except terms are introduced hierarchically: sets of terms $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$ are each examined greedily, where the overall strongest model of one tier forms the seed model for the subsequent tier. This is depicted in the main text in **??**. A corresponding ES is given as follows.

```
class ExampleGreedySearchTiered(
    exploration_strategy.ExplorationStrategy
):
    r"""
    Greedy search in tiers.

    Terms are batched together in tiers;
    tiers are searched greedily;
    a single tier champion is elevated to the subsequent tier.

    """

    def __init__(
        self,
        exploration_rules,
        **kwargs
    ):
        super().__init__(
            exploration_rules=exploration_rules,
            **kwargs
        )
        self.true_model = 'pauliSet_1_x_d3+pauliSet_1J2_yJy_d3+
           pauliSet_1J2J3_zJzJz_d3'
        self.initial_models = None
        self.term_tiers = {
            1 : ['pauliSet_1_x_d3', 'pauliSet_1_y_d3', '
               pauliSet_1_z_d3' ],
            2 : ['pauliSet_1J2_xJx_d3', 'pauliSet_1J2_yJy_d3', '
               pauliSet_1J2_zJz_d3'],
            3 : ['pauliSet_1J2J3_xJxJx_d3', '
               pauliSet_1J2J3_yJyJy_d3', 'pauliSet_1J2J3_zJzJz_d3
               '],
```

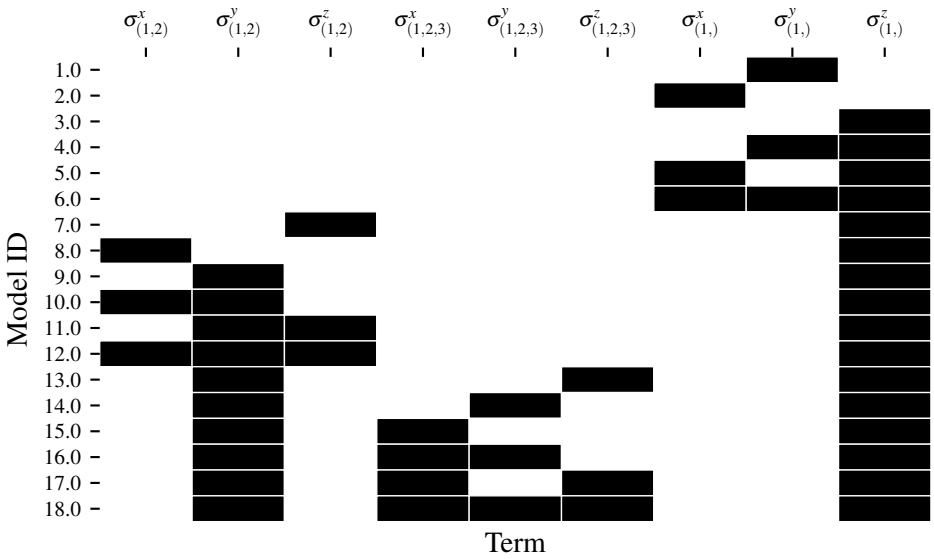
```
        }
        self.tier = 1
        self.max_tier = max(self.term_tiers)
        self.tier_branch_champs = {k : [] for k in self.
            term_tiers}
        self.tier_champs = {}
        self.prune_completed_initially = True
        self.check_champion_reducibility = True

    def generate_models(
        self,
        model_list,
        **kwargs
    ):
        self.log_print([
            "Generating models in tiered greedy search at spawn
                step {}.".format(
                self.spawn_step,
            )
        ])

        if self.spawn_stage[-1] is None:
            try:
                previous_branch_champ = model_list[0]
                self.tier_branch_champs[self.tier].append(
                    previous_branch_champ)
            except:
                previous_branch_champ = None

        elif "getting_tier_champ" in self.spawn_stage[-1]:
            previous_branch_champ = model_list[0]
            self.log_print([
                "Tier champ for {} is {}".format(self.tier,
                    model_list[0])
            ])
            self.tier_champs[self.tier] = model_list[0]
            self.tier += 1
            self.log_print(["Tier now = ", self.tier])
            self.spawn_stage.append(None) # normal processing
```

```
            if self.tier > self.max_tier:
                self.log_print(["Completed tree for ES"])
                self.spawn_stage.append('Complete')
                return list(self.tier_champs.values())
        else:
            self.log_print([
                "Spawn stage:", self.spawn_stage
            ])

        new_models = greedy_add(
            current_model = previous_branch_champ,
            terms = self.term_tiers[self.tier]
        )
        self.log_print([
            "tiered search new_models=", new_models
        ])

        if len(new_models) == 0:
            # no models left to find - get champions of branches
               from this tier
            new_models = self.tier_branch_champs[self.tier]
            self.log_print([
                "tier champions: {}".format(new_models)
            ])
            self.spawn_stage.append("getting_tier_champ_{}".
               format(self.tier))
        return new_models

    def check_tree_completed(
        self,
        spawn_step,
        **kwargs
    ):
        r"""
        QMLA asks the exploration tree whether it has finished
           growing;
        the exploration tree queries the exploration strategy
           through this method
        """
        if self.tree_completed_initially:
```

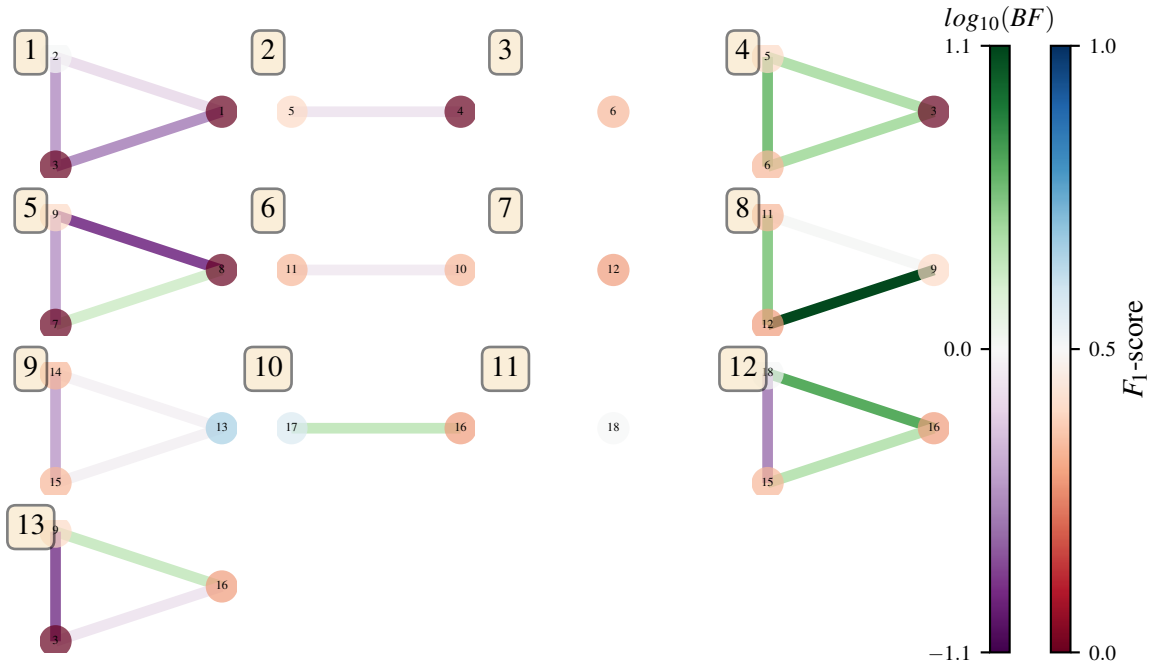
```
            return True
        elif self.spawn_stage[-1] == "Complete":
            return True
        else:
            return False


def greedy_add(
    current_model,
    terms,
):
    r"""
    Combines given model with all terms from a set.

    Determines which terms are not yet present in the model,
    and adds them each separately to the current model.

    :param str current_model: base model
    :param list terms: list of strings of terms which are to be
       added greedily.
    """

    try:
        present_terms = current_model.split('+')
    except:
        present_terms = []
    nonpresent_terms = list(set(terms) - set(present_terms))

    term_sets = [
        present_terms+[t] for t in nonpresent_terms
    ]

    new_models = ["+".join(term_set) for term_set in term_sets]

    return new_models
```

Listing C.18: ExampleGreedySearchTiered exploration stategy

with corresponding results in Fig. C.8.



(a)



(b)

Figure C.8: Tiered greedy exploration strategy. **(a)**, composition_of_models. **b**, graphs_of_branches_ExampleGreedySearchTiered: shows which models reside on each branches of the exploration tree. Models are coloured by their F_1-score, and edges represent the BF between models. In each tier, three branches greedily add terms, and a fourth branch considers the champions of the first three branches in order to nominate a tier champion. The final branch consists only of the tier champions, to nominate the global champion, $\hat{H}'$.

BIBLIOGRAPHY

[1] The Difference Between AI and Machine Learning, Jan 2021. [Online; accessed 7. Jan. 2021].

[2] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[3] Alan M Turing. Computing machinery and intelligence. In *Parsing the turing test*, pages 23–65. Springer, 2009.

[4] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[5] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[8] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[9] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168, 2006.

[10] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160(1):3–24, 2007.

[11] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[12] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, 1992.

[13] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[14] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 21–35. Springer, 1996.

[15] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[16] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

[17] Jeremy Adcock, Euan Allen, Matthew Day, Stefan Frick, Janna Hinchliff, Mack Johnson, Sam Morley-Short, Sam Pallister, Alasdair Price, and Stasja Stanisic. Advances in quantum machine learning. *arXiv preprint arXiv:1512.02900*, 2015.

[18] Ewin Tang. A quantum-inspired classical algorithm for recommendation systems. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 217–228, 2019.

[19] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. The quest for a quantum neural network. *Quantum Information Processing*, 13(11):2567–2586, 2014.

[20] Yudong Cao, Gian Giacomo Guerreschi, and Alán Aspuru-Guzik. Quantum neuron: an elementary building block for machine learning on quantum computers. *arXiv preprint arXiv:1711.11240*, 2017.

[21] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. Quantum principal component analysis. *Nature Physics*, 10(9):631–633, 2014.

[22] Giuseppe Carleo and Matthias Troyer. Solving the quantum many-body problem with artificial neural networks. *Science*, 355(6325):602–606, 2017.

[23] Giacomo Torlai, Guglielmo Mazzola, Juan Carrasquilla, Matthias Troyer, Roger Melko, and Giuseppe Carleo. Neural-network quantum state tomography. *Nature Physics*, 14(5):447–450, 2018.

[24] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5:4213, 2014.

[25] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[26] Kenneth De Jong. Evolutionary computation: a unified approach. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 327–342, 2020.

[27] Sean Luke. 1 essentials of metaheuristicsl. 1.

[28] Lothar M Schmitt. Theory of genetic algorithms. *Theoretical Computer Science*, 259(1-2):1–61, 2001.

[29] Brian Flynn. Mathematical introduction to quantum computation, 2015. Undergraduate thesis.

[30] Quantum model learning agent documentation. https://quantum-model-learning-agent.readthedocs.io/en/latest/, Jan 2021. [Online; accessed 12. Jan. 2021].

[31] Brian Flynn. Quantum model learning agent. https://github.com/flynnbr11/QMLA, 2021.