Quantum machine learning: on the design, trainability and noise-robustness of near-term algorithms

Skolik, A.

CHAPTER 3

Machine learning

The field of machine learning (ML) is concerned with developing systems that learn to perform certain tasks without being explicitly programmed to do so. The methods used in this field encompass a wide range of techniques, ranging from simple linear regression to more complex approaches like neural networks [83, 84]. How the problem of learning a certain task is addressed can be broadly separated into three branches: supervised learning, unsupervised learning, and reinforcement learning. What all three of these methods have in common is that they are based on a *model*, the trainable part of the algorithm that will later execute the learned task. In the supervised setting, a model is trained based on a set of labeled data samples, to then label samples not seen during training. In unsupervised learning, there are no labels assigned to the training data, but the goal of the algorithm is to infer the underlying structure of the data, e.g., by sorting it into separate clusters. In reinforcement learning, there is no training data per se, but the algorithm learns in a trial-and-error fashion by interacting with an environment.

Depending on which type of learning is performed, one can also choose between a number of different models. In this thesis, we will study ML from the neural network perspective [85], as these types of models are most closely related to the VQAs described in Section 2.2.1. We will first describe neural networks and their training in Section 3.1, and then move on to introduce reinforcement learning in more depth in Section 3.2, which is the learning algorithm we mainly focus on in this thesis. Finally, we outline the intersection of machine learning and quantum computing, referred to as *quantum machine learning*, in Section 3.3.

3.1 Neural networks

3.1.1 Neurons, layers, and backpropagation

In its simplest form, a neural network (NN) consists of layers of artificial neurons that are loosely based on biological neurons, where the output of the k-th neuron, given an input vector $\boldsymbol{x}$ of length m, is of the form

$$f_k(\boldsymbol{x}) = \sigma\left(\sum_{j=0}^{m} w_{kj} x_j\right), \tag{3.1}$$

where σ denotes an *activation function* that serves the purpose of introducing nonlinearities to the NN. The w_{kj} represent trainable weights, which are optimized such that the whole network gives the desired output on a given input, similarly to the parameters $\boldsymbol{\theta}$ in the VQAs we introduced in Section 2.2.1. When NNs are trained with gradient descent, the activation function needs to be differentiable or at least allow the computation of subderivatives in order to compute gradients for the weight updates. We denote the weights between layers l and $l-1$ as $\boldsymbol{w^{(l)}}$, and the function that represents the action of the full layer as $\sigma^{(l)}$. Each layer in a NN can theoretically contain arbitrarily many neurons, where the input of the neurons in each layer is given by the output of the neurons in the previous layer. When a higher number of layers is involved, this is commonly referred to as *deep learning*. The full network with input $\boldsymbol{x}$ can then be written as the following composition of functions starting from the last layer L,

$$f(\boldsymbol{x}) = \sigma^{(L)}(\boldsymbol{w^{(L)}}\sigma^{L-1}(\boldsymbol{w^{(L-1)}}\sigma^{L-2}(\ \ldots\ \boldsymbol{w^{1}}\sigma^{0}(\boldsymbol{w^{0}x})\ \ldots\ ))). \tag{3.2}$$

A visualization of this type of NN can be seen in Figure 3.1. The above equation represents the simplest form of a NN, where the neurons in each layer are only connected to the neurons in neighboring layers, called a *feedforward* NN. The training data set consists of pairs of examples $\mathcal{X} = \{(x_i, y_i)\}$, and the goal is to adjust the weights of the network such that for each x_i, the network produces the desired output y_i. How close the model output $\hat{y}_i$ is to the target output, also called the *loss*, is quantified in terms of a *cost function*,

$$C(y_i, \hat{y}_i). \tag{3.3}$$

To optimize the weights $\boldsymbol{w}$, typically gradient descent with the *backpropagation algorithm* is used [86, 87]. This algorithm provides an efficient method to compute

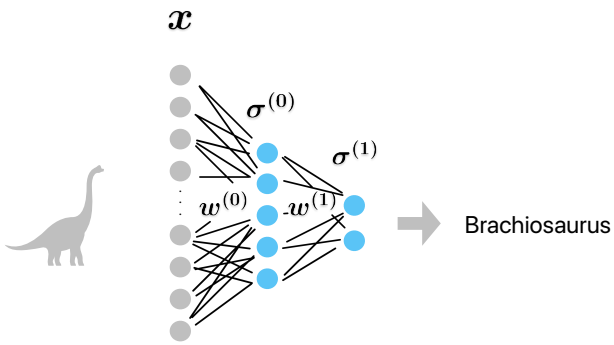

Figure 3.1: Depiction of a neural network that takes images of dinosaurs as input, and classifies them into two categories. The $\boldsymbol{w^{(l)}}$ are trainable weights, the $\sigma^{(l)}$ are layers of neurons with activation functions, and $\boldsymbol{x}$ is an input vector. Black lines show weights $w_{kj}^{(l)}$ between two neurons in adjacent layers.

gradients in NNs, utilizing their layered structure. Naively, one can compute the gradient of the cost function w.r.t. a given weight $w_{kj}^{(l)}$ via the chain rule. However, doing this will require redundant computations of partial derivatives as we work through the layers. Due to the fact that each layer only depends on its successors by how they affect the cost function, and does so in a linear manner, the computation of the gradient can be regarded layer-by-layer and performed without re-computing redundant derivatives. More formally, the loss of a NN with L layers on a pair of samples (x_i, y_i) is

$$C(y_i, \sigma^{(L)}(\boldsymbol{w^{(L)}}\sigma^{(L-1)}(\boldsymbol{w^{(L-1)}}\sigma^{(L-2)}(\ \ldots\ \boldsymbol{w^{(1)}}\sigma^{(0)}(\boldsymbol{w^{(0)}}\boldsymbol{x})\ \ldots\ )))). \tag{3.4}$$

The gradient of the cost function given input x is then the following,

$$\nabla_x C = \boldsymbol{w^{(0)T}} \cdot (\sigma^{(0)})' \circ \cdots \circ \boldsymbol{w^{(L-1)T}} \cdot (\sigma^{(L-1)})' \circ \boldsymbol{w^{(L)T}} \cdot (\sigma^{(L)})' \circ \nabla_{a^{(L)}} C, \tag{3.5}$$

where we denote the output of layer l as $a^{(l)}$, and the derivative of $\sigma^{(l)}$ as $(\sigma^{(l)})'$. Instead of computing the derivatives from left to right, where each computation at the i-th layer includes derivatives of the following layers of nested functions, in the backpropagation algorithm, derivatives are evaluated right to left, and the $a^{(l)}$ as well as the derivatives of each layer are cached along the way. In addition, propagating the error of a given training data pair backwards through the network constitutes multiplication of the vector of partial products of the derivatives with a matrix of weights for each layer. Performing the computation in the opposite

direction, on the other hand, constitutes of a multiplication of two matrices at each layer. These two changes in the procedure of computing gradients make the backpropagation algorithm much more efficient than a naive calculation that includes all the $L - i$ terms in every step, and is key to enable training large-scale NNs with billions of parameters. This is different to gradient computation in the VQAs we described in Section 2.2.1. There, gradients are obtained by the parameter-shift rule in Equation (2.16), which requires two circuit evaluations per trainable gate in the circuit. In particular, generic quantum circuits do not allow for a layer-wise computation of partial derivatives as in the backpropagation algorithm, where derivatives previously computed for other parameters in the circuit can be reused.

As mentioned before, the above describes computation of gradients in a feedforward NN, which is the simplest NN in terms of the connections between neurons. While gradients in this architecture can be computed efficiently, the all-to-all connectivity between each of the layers poses other problems for the trainability of the network. The most infamous of these problems is that of *vanishing* or *exploding gradients* [88]. This is a consequence of the layer-by-layer computation of gradients in the backpropagation algorithm. The gradient is computed by multiplying partial derivatives for each layer, so when derivatives are small, and in particular when they are smaller than one, multiplying several of these values will lead to a gradient that is vanishing exponentially in the number of terms that are multiplied. A similar effect is present when partial derivatives are large: multiplying increasingly large numbers in each layer leads to a blow-up of the derivative terms. In order to overcome this problem, weight initialization strategies [89], as well as activation functions [90] and more elaborate NN architectures [91] have been developed, and research in this area is still ongoing.

Despite these practical hurdles, NNs have become the most popular models used for large-scale ML due to their flexibility and the broad range of tasks they can be applied to. Indeed, theory shows that NNs can, in principle, approximate any function, a result that is known as the *universal approximation theorem* [92, 93]. There are actually a number of these universal approximation theorems, each pertaining different types of NN architectures: the *arbitrary width* case concerns networks with only one layer that contains many neurons [92, 94, 95], the *arbitrary depth* case is about networks that contain a limited number of neurons in each layer, but use several of those layers [96], and the *bounded width and depth* case is a

combination of the first two cases [97]. These results show that theoretically, NNs can represent many functions of interest, given a suitable set of weights. However, these theorems neither tell us anything about how to obtain these weights, nor how hard it is to find them. In order to make full use of the power of NNs, several challenges in high-dimensional optimization as the one mentioned above have to be overcome. In the following section, we will encounter one of those challenges, that results from a fundamental tradeoff between model complexity, and a model's ability to perform well on previously unseen data.

3.1.2 Generalization and overfitting

As briefly alluded to in the introduction of this chapter, the main goal of a machine learning algorithm is to learn to perform a given task without being explicitly programmed to do so. Information is inferred from a set of data samples or interactions with an environment, in order to use this information to process previously unseen data. In machine learning literature, the ability to use this information on unfamiliar data is referred to as *generalization*, and the failure to do so as the *generalization error*. Usually, these terms are formally defined in a supervised learning setting, where the algorithm is given a set of training data points $\mathcal{X} = \{(X = x_i, Y = y_i)\}$ sampled from some joint distribution of $X \times Y$, and the goal is to produce the output y_i given the input x_i, and the error between model output $\hat{y}_i$ and target label y_i is given by the cost function $C(y_i, \hat{y}_i)$, as in Section 3.1.1. The average difference between the output of a model parametrized by $\boldsymbol{\theta}$, denoted $\hat{y}_i^{(\boldsymbol{\theta})}$, and the target labels for a training data set of size N is then given as

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} C(y_i, \hat{y}_i^{(\boldsymbol{\theta})}) \tag{3.6}$$

and is called the *empirical risk*. Naively, one could now define the goal of the ML model as finding the set of parameters $\boldsymbol{\theta}^*$ that minimizes the empirical risk on the given training data set,

$$\boldsymbol{\theta}^* = \text{argmin}_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}). \tag{3.7}$$

However, even under the assumption that there exists a set of parameters that yields a model with an empirical risk of zero, it is not clear that these parameters are also optimal for samples that were not included in the training data, i.e., whether this set of optimal parameters generalizes to unseen samples. Even worse, the parameters that are optimal for the training data may lead to higher error on

data not included in the training set, as the model is now overly fine-tuned on the training data. This effect is commonly referred to as *overfitting*.

To quantify the generalization performance of a model, it is helpful to specify the data that it was trained on in the definition of the risk,

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{X}) = \frac{1}{|\mathcal{X}|} \sum_{i=1}^{|\mathcal{X}|} C(y_i, \hat{y}_i^{(\boldsymbol{\theta})}). \tag{3.8}$$

Now, assuming that we have access to the probability distribution that generated the training data, $p^*(x, y)$, we can define the *population risk*, also called *total risk*, as the theoretical expected loss

$$\mathcal{L}(\boldsymbol{\theta}; p^*) = \mathbb{E}_{p^*(x,y)}[C(y_i, \hat{y}_i^{(\boldsymbol{\theta})})]. \tag{3.9}$$

In practice, we usually do not have access to p^*, however, it lets us define the *generalization gap* as the difference between the population and the empirical risk, $\mathcal{L}(\boldsymbol{\theta}; p^*) - \mathcal{L}(\boldsymbol{\theta}; \mathcal{X})$. A large generalization gap, meaning that the empirical risk is low while the population risk is high, is a clear sign for overfitting. But how can we prevent the model from overfitting without knowing the population risk?

An empirical approach to prevent overfitting on the training data is to split the available data into three partitions: i) the training data that is used to fit the model, ii) a validation set that is used for hyperparameter tuning, and iii) a test set that is finally used to evaluate model performance. Further splitting the data into validation and test set is beneficial, as most ML models' performance strongly depends on the hyperparameter setting used. Hyperparameters in a NN are, for example, the learning rate η used to determine the step size of the gradient-based updates of weights, or the number of layers and neurons in each layer in the network. Once a well-performing set of hyperparameters is found based on the validation data set, the final model performance is evaluated on the test data set, which was not used in any of the previous steps of the model training and selection. There are many different techniques to perform this division of the data set that facilitate making the best use of the limited data that is available to train and evaluate a model. A commonly used technique is called *k-fold cross validation*, where the full data set is split into k parts, and the training and evaluation of a model is repeated multiple times where different partitions of the data act as training, validation and test sets, respectively.

Preventing overfitting in ML models is a highly non-trivial task, and a variety of different approaches have been developed for this. A models' capacity for overfitting is closely related to its complexity, where a rule of thumb is that the more complex a model is, the closer it can and will fit the training data exactly, unless measures to prevent this are taken. This results in a tradeoff between the model complexity and its generalization performance, where both, models that are too complex or not complex enough, will have bad generalization performance and one is required to find the model that has *just the right* level of complexity for a given learning task. This is referred to as the *bias-variance tradeoff* [98]. To make this formal, we consider the *bias-variance decomposition* of the expected error $\mathbb{E}_{p^*(x,y)}[(\hat{y} - y)^2|x]$, where for ease of notation we drop the explicit reference to $p^*(x, y)$ and x in the following, assuming that the cost function is the mean squared error between model prediction $\hat{y}$ and true label y,

$$\begin{aligned}
\mathbb{E}_{p^*(x,y)}[(\hat{y} - y)^2|x] &= \mathbb{E}[\hat{y}^2 - 2y\hat{y} + y^2] \\
&= \hat{y}^2 - 2\hat{y}\mathbb{E}[y] + \mathbb{E}[y^2] \\
&= \hat{y}^2 - 2\hat{y}\mathbb{E}[y] + \mathbb{E}[y]^2 + \mathrm{Var}[y] \\
&= (\hat{y} - \mathbb{E}[y])^2 + \mathrm{Var}[y] \\
&\triangleq (\hat{y} - y^*) + \mathrm{Var}[y],
\end{aligned} \tag{3.10}$$

where $y^* = \mathbb{E}[y]$ is the optimal prediction given x. If we now treat $\hat{y}$ as a random variable, where we repeatedly sample a data set from p^*, train the model, and generate predictions $\hat{y}$, we get the expected error

$$\begin{aligned}
\mathbb{E}[(\hat{y} - y)^2] &= \mathbb{E}[(\hat{y} - y^*)^2] + \mathrm{Var}[y] \\
&= \mathbb{E}[(y^*)^2 - 2y^*\hat{y} + \hat{y}^2] + \mathrm{Var}[y] \\
&= (y^*)^2 - 2y^*\mathbb{E}[\hat{y}] + \mathbb{E}[\hat{y}^2] + \mathrm{Var}[y] \\
&= (y^*)^2 - 2y^*\mathbb{E}[\hat{y}] + \mathbb{E}[\hat{y}]^2 + \mathrm{Var}[\hat{y}] + \mathrm{Var}[y] \\
&= \underbrace{(y^* - \mathbb{E}[\hat{y}])^2}_{\text{bias}} + \underbrace{\mathrm{Var}[\hat{y}]}_{\text{variance}} + \underbrace{\mathrm{Var}[y]}_{\text{Bayes error}}.
\end{aligned} \tag{3.11}$$

The bias term in Equation (3.11) represents the average error of the model predictions, while the second term informs us about the variance of model predictions on the training data. The third term is the variance over the true labels that we have no control over, and can therefore ignore in this discussion. High variance hints at an increased sensitivity of the model to small fluctuations in the training data, which can be a result of the model fitting the training data too closely, and

is therefore a sign of overfitting. The bias of the model results from erroneous predictions when the model misses crucial information in modeling the input-output relation, and is therefore a sign of *underfitting*. The above tells us that restricting the model by introducing a bias can actually be beneficial, as long as it reduces the variance. As described above, one difficulty at the heart of ML lies in finding an adequate balance between these two terms for the learning task at hand. In the following section, we will see how we can deliberately introduce an inductive bias based on knowledge about the training data into the model, in order to increase model performance and improve generalization.

3.1.3 Geometric deep learning

In the past years, the dimensionality of the input data of problems that are addressed with deep learning has kept increasing in size, and state-of-the-art NNs now contain billions of trainable weights. Finding a good set of weights poses an extremely high-dimensional optimization problem that comes with its own challenges. Most notably, we face the *curse of dimensionality* [99]: the data required to solve these high-dimensional learning problems often grows exponentially with the dimensionality. This means that learning about generic, unstructured data in high-dimensional spaces is generally infeasible (an effect we have already encountered for quantum models in Section 2.2.1.2). However, most of the problems that we are interested in solving with ML do have an underlying structure. If this knowledge about the structure of the learning task can be imposed on the model itself before training, this effectively reduces the dimensionality of the optimization problem. This problem-dependent imposing of structure is also referred to as the *inductive bias* of a model.

The field of *geometric deep learning* [100] is concerned with endowing models with an inductive bias from the perspective of geometry and symmetries. Most of the data that we are interested in learning about is generated by a process in our physical world, and many of these physical processes have underlying symmetries that make reasoning about them more easy. In fact, the study of symmetries is one of the cornerstones of modern physics, and underlies important theories like that of general relativity [101]. Therefore, it is not surprising that learning of functions in high-dimensional spaces can also be simplified by utilizing the symmetries that are present in the given training data. A key example of a NN with a problem-dependent inductive bias is the convolutional NN, that is used to process images

[102, 103, 104]. One symmetry present in images is that of translation invariance, i.e., shifting an object around in an image does not change the object itself. For example, a model that is used for object tracking in video data should recognize that a ball that is thrown and will therefore appear in different locations of successive frames of the video, is indeed the same ball. The layers of convolutional NNs are designed precisely so that they are invariant to translations. Another common data structure that has a corresponding symmetry-preserving NN architecture is that of graphs [105]. Graph NNs preserve permutation invariance, or the closely related permutation equivariance, of graph nodes, meaning that they are not affected by the order in which representations of graph nodes are fed into the network. Graph NNs can be used for many different types of learning tasks, like predicting properties of molecules [106], inferring relationships in social networks [107], or solving instances of combinatorial optimization problems [108]. The above types of geometric models have played a key role in enabling recent breakthroughs in deep learning, alongside the backpropagation algorithm we described in Section 3.1.1. In Chapter 6 of this thesis, we will go into more detail about geometric learning, and introduce a quantum model that is equivariant under permutation of graph nodes, similarly to the graph NNs described above.

3.2 Reinforcement learning

In this section, we focus on the branch of machine learning that is considered in most of the following chapters of this thesis: reinforcement learning (RL). In RL, an agent does not learn from a fixed data set as in other types of learning, but by making observations on and interacting with an environment [109]. This distinguishes it from the other two main branches of ML, supervised and unsupervised learning, and each of the three comes with its individual challenges. In a supervised setting, an agent is given a fixed set of training data that is provided with the correct labels, where difficulties arise mainly in creating models that do not overfit the training data and keep their performance high on unseen samples. In unsupervised learning, training data is not labeled and the model needs to discover the underlying structure of a given data set, and the challenge lies in finding suitable loss functions and training methods that enable this. RL also comes with a number of challenges: there is no fixed set of training data, but the agent generates its own samples by interacting with an environment. These samples are not labeled, but only come with feedback in form of a reward. Additionally, the training data keeps changing

throughout the learning process, as the agent constantly receives feedback from interacting with its environment. This often results in a high number of interactions that is required to successfully learn in a given environment, making RL the most sample-inefficient method out of the three described approaches.

An environment consists of a set of possible states $\mathcal{S}$ that it can take, and a set of actions $\mathcal{A}$ which the agent can perform to alter the environment's state. Both state and action spaces can be continuous or discrete. The function that models the agent's behavior in the environment is called the *policy* $\pi(a|s)$, which gives the probability of taking action a in a given state s. An agent interacts with an environment by performing an action a_t at time step t in state s_t, upon which it receives a reward r_{t+1}. A tuple (s, a, r, s') of these four quantities is called a *transition*, and a sequence of transitions is a *trajectory*. Another ingredient in the interaction between agent and environment are the transition probabilities from state s to s' after performing action a. They are represented by the *transition function* $P^a_{ss'}$, which for environments that can be described in terms of a Markov decision process (MDP) is defined as follows,

$$P^a_{ss'} = P(s'|s, a). \tag{3.12}$$

For environments that are not MDPs, the transition probabilities may depend on the whole trajectory instead of just the previous state and action. The transition function is a property of the environment, and one can distinguish between *model-free* algorithms as the ones we study in this work, where transition probabilities are not learned, and *model-based* environments which learn the transition function as a model of the environment. The quality of the agent's actions in the environment is evaluated by a *reward function* that is tailored to the learning task at hand, and the agent's goal is to learn a policy $\pi(a|s)$ that maximizes its total reward. The expected reward over a sequence of time steps starting at t is called the *return*,

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{3.13}$$

where $\gamma \in [0, 1)$ is a discount factor introduced to prevent divergence of the infinite sum. The return G_t should be viewed as the agent's expected reward when starting from time step t and summing the discounted rewards of potentially infinitely many future time steps, where maximizing the return at step t implies also maximizing the return of future time steps. Note that the task is to maximize an expected

value, and that the reward r_t in Equation (3.13), and therefore G_t are random variables. Environments often naturally break down into so-called *episodes*, where the sum in Equation (3.13) is not infinite, but only runs over a fixed number of steps called *horizon* H. An example of this are environments based on games, where one episode comprises one game played and an agent learns by playing a number of games in series.

Much of the theoretical work on RL is on so-called *tabular* algorithms, where the rewards for possible transitions are stored in a table. While this simplifies certain theoretical analyses such as proofs of convergence [110], these types of algorithms quickly become inefficient with growing state and action spaces. Therefore, different methods based on *function approximation* have been developed, where rewards are not stored explicitly anymore but approximately represented by a function. One example of RL with function approximation is deep reinforcement learning (DRL), where the function approximator is a NN. Since the advent of NNs in the past decades, much of RL research has focused on these types of algorithms, and we also study RL with function approximation in this thesis. For this reason, we only describe the most important concepts and techniques for RL with function approximators here, as a general introduction to the broad field of RL is out of the scope of this thesis. For more information on the history of RL, its connection to dynamic programming and the Bellman equation [111], and the various theoretical and practical formalisms developed in this field, the interested reader can refer to [109].

3.2.1 Value-based and policy-based learning

RL algorithms can be categorized into *value-based* and *policy-based* learning methods. Both approaches aim to maximize the return as explained above, but use different figures of merit to achieve this. Both approaches also have their disadvantages as we will see below, and which type of algorithm should be used depends on the environment at hand. The main difference between the two approaches is how the policy is realized. In general, performance is evaluated based on a state-value function $V_\pi(s)$,

$$V_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s], \tag{3.14}$$

which is the expected return when following policy π starting from state s at initial time step t, and the goal of a RL algorithm is to learn the optimal policy π_* which maximizes the expected return for each state.

A policy-based algorithm seeks to learn an optimal policy directly, that is, learn a probability distribution of actions given states. In this setting, the policy is implemented in form of a parametrized conditional probability distribution $\pi(a|s;\boldsymbol{\theta})$, and the goal of the algorithm is to find parameters $\boldsymbol{\theta}$ such that the resulting policy is optimal. The figure of merit in this setting is some performance measure $J(\boldsymbol{\theta})$ that we seek to maximize, that in the fixed-horizon setting is equal to the value function in Equation (3.14),

$$J(\boldsymbol{\theta}) := V_{\pi_{\boldsymbol{\theta}}}(s). \tag{3.15}$$

This performance measure is used to find a good policy, however, once the policy has been learned $J(\boldsymbol{\theta})$ is not required for action generation. Typically, these algorithms perform gradient ascent on an approximation of the gradient of the performance measure $\nabla J(\boldsymbol{\theta})$, which is obtained by Monte Carlo samples of policy rollouts (i.e., a set of observed interactions with the environment performed under the given policy), and are hence called *policy gradient* methods. This approach produces smooth updates on the policy (as opposed to value-based algorithms, where a small change in the value function can drastically alter the policy) that enable proofs of convergence to locally optimal policies [112]. However, it also suffers from high variance as updates are purely based on Monte Carlo samples of interactions with the environment [113]. A number of methods to reduce this variance have been developed, like adding a value-based component as described below to a policy-based learner in the so-called *actor-critic* method [114].

In a value-based algorithm, a value function as in Equation (3.14) is learned instead of the policy. The policy is then implicitly given by the value function: an agent will pick the action which yields the highest expected return according to $V_\pi(s)$. A concrete example of value-based learning is given in Section 3.2.2, where we describe the *Q-learning* algorithm that we use in later chapters of this thesis. While value-based algorithms do not suffer from high training variance as policy gradient learning does, they often require more episodes to converge. They also result in deterministic policies, as the agent always picks the action that corresponds to the highest expected reward, so this approach will fail when the optimal policy is stochastic and post-training action selection is performed according to the argmax policy.[1] Additionally, the policy resulting from a parametrized value function can change substantially after a single parameter update (i.e., a very small change in

[1] Consider for example a game of poker where bluffing is a valid action to scare other players into folding, but quickly becomes obvious when greedily done in every round.

the value function can lead to picking a different action after an update). This results in theoretical difficulties to prove convergence when a function approximator is used to parametrize the value function, hence there are even fewer theoretical guarantees for this approach than for policy gradient methods. On the other hand, it was the advent of Q-learning with function approximation that made it possible to solve extremely complex problems such as Go with a RL approach [29].

Both approaches have their own (dis-) advantages, and while the popularity of either method has surpassed the other at some point in the last decades, there is no clear winner. As mentioned above, an actor-critic approach combines a policy-based and value-based learner to leverage the advantages of both while alleviating the disadvantages, and this method is among the stat-of-the-art in classical RL literature [115]. Additionally, it can be easier to learn either the policy or the value function depending on a given environment. For this reason, both approaches are worth being studied independently.

3.2.2 Q-learning

In Q-learning, we are not interested in the state-value function as shown in Equation (3.14), but in the closely related action-value function $Q_\pi(s, a)$,

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a], \tag{3.16}$$

which also gives us the expected return assuming we follow a policy π, but now additionally conditioned on an action a. We call the optimal Q-function for all state-action pairs $Q_*(s, a) = \max_\pi Q_\pi(s, a)$, and an optimal policy can be easily derived from the optimal values by taking the highest-valued action in each step,

$$\pi_*(a|s) = \underset{a}{\operatorname{argmax}}\ Q_*(s, a). \tag{3.17}$$

The goal in Q-learning is to learn an estimate, $Q(s, a)$, of the optimal Q-function. In its original form, Q-learning is a tabular learning algorithm, where a so-called *Q-table* stores Q-values for each possible state-action pair [116]. When interacting with an environment, an agent chooses its next action depending on the Q-values as

$$a_t = \underset{a}{\operatorname{argmax}}\ Q(s_t, a), \tag{3.18}$$

where a higher value designates a higher expected reward when action a is taken in state s_t as opposed to the other available actions. When we consider learning

by interaction with an environment, it is important that the agent is exposed to a variety of transitions to sufficiently explore the state and action space. Intuitively, this provides the agent with enough information to tell apart good and bad actions given certain states. Theoretically, visiting all state-action transitions infinitely often is one of the conditions that are required to hold for convergence proofs of tabular Q-learning to an optimal policy [110]. Clearly, if we always follow an argmax policy, the agent may only get access to a limited part of the state and action space. To ensure sufficient exploration in a Q-learning setting, a so-called *ϵ-greedy policy* is used. That is, with probability ϵ a random action is performed and with probability $1 - \epsilon$ the agent chooses the action which corresponds to the highest Q-value for the given state as in Equation (3.18). Note that the ϵ-greedy policy is only used to introduce randomness to the actions picked by the agent during training, but once training is finished, a deterministic argmax policy is followed.

The Q-values are updated with observations made in the environment by the following update rule,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)], \tag{3.19}$$

where α is a learning rate, r_{t+1} is the reward at time $t + 1$, and γ is a discount factor. Intuitively, this update rule provides direct feedback from the environment in form of the observed reward, while simultaneously incorporating the agent's own expectation of future rewards at the present time step via the maximum achievable expected return in state s_{t+1}. In the limit of visiting all (s, a) pairs infinitely often, this update rule is proven to converge to the optimal Q-values in the tabular case [110].

Obviously, the tabular approach is intractable for large state and action spaces. For this reason, the Q-table was replaced in subsequent work by a Q-function approximator which does not store all Q-values individually [117, 118]. In the landmark work [119], the authors use a NN as the Q-function approximator which they call *deep Q-network* (DQN) and the resulting algorithm the *DQN algorithm*, and demonstrate that this algorithm achieves human-level performance on a number of arcade games. In this work, the agent chooses actions based on an ϵ-greedy policy as described above. Typically ϵ is chosen large in the beginning and then

decayed in subsequent iterations, to ensure that the agent can sufficiently explore the environment at early stages of training by being exposed to a variety of states. The authors of [119] also utilize two other methods to stabilize training these models: (i) *experience replay*: past transitions and their outcomes are stored in a memory, and the batches of these transitions that are used to compute parameter updates are sampled at random from this memory to remove temporal correlations between transitions, (ii) adding a second so-called *target network* to compute the expected Q-values for the update rule, where the target network has an identical structure as the DQN, but is not trained and only sporadically updated with a copy of the parameters of the DQN. Note that the target network is only used for computing parameter updates, but is not needed after the training procedure where only the Q-network is used. We refer to the original paper for further details on the necessity of these techniques to stabilize training of the DQN algorithm.

The DQN is then trained almost in a supervised fashion, where the training data and labels are generated by the DQN itself through interaction with the environment. At each update step, a batch $\mathcal{B}$ of previous transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ is chosen from the replay memory. To perform a model update, we need to compute $\max_a Q(s_{t+1}, a)$. When we use a target network, this value is not computed by the DQN, but by the target network $\hat{Q}$. To make training more efficient, in practice the Q-function approximator is redefined as a function of a state parametrized by $\boldsymbol{\theta}$, $Q_{\boldsymbol{\theta}}(s) = \boldsymbol{q}$, which returns a vector of Q-values for all possible actions instead of computing each $Q(s, a)$ individually. We now want to perform a supervised update of $Q_{\boldsymbol{\theta}}$, where the label is obtained by applying the update rule in Equation (3.20) to the DQN's output. To compute the label for a state s that we have taken action a on in the past, we take a copy of $Q_{\boldsymbol{\theta}}(s)$ which we call $\boldsymbol{q_\delta}$, and only the ith entry of $\boldsymbol{q_\delta}$ is altered where i corresponds to the index of the action a, and all other values remain unchanged. The estimated maximum Q-value for the following state s_{t+1} is computed by $\hat{Q}_{\boldsymbol{\theta}_\delta}$, and the update rule for the i-th entry in $\boldsymbol{q_\delta}$ takes the following form

$$\boldsymbol{q}_{\delta_i} = r_{t+1} + \gamma \cdot \max_a \hat{Q}_{\boldsymbol{\theta}_\delta}(s_{t+1}, a), \tag{3.20}$$

where $\boldsymbol{\theta}_\delta$ is a periodically updated copy of $\boldsymbol{\theta}$. The loss function $\mathcal{L}$ is the mean squared error (MSE) between $\boldsymbol{q}$ and $\boldsymbol{q}_\delta$ on a batch of sample transitions $\mathcal{B}$,

$$\mathcal{L}(\boldsymbol{q}, \boldsymbol{q}_\delta) = \frac{1}{|\mathcal{B}|} \sum_{b \in \mathcal{B}} (\boldsymbol{q}_b - \boldsymbol{q}_{\delta_b})^2. \tag{3.21}$$

Note that because $\boldsymbol{q}_\delta$ is a copy of $\boldsymbol{q}$ where only the i-th element is altered via the update rule in Equation (3.20), the difference between all other entries in those two vectors is zero. As Q-values are defined in terms of (s, a)-pairs, this approach does not naturally apply to environments with continuous action spaces. In this case, the continuous action space has to be binned into a discrete representation.

Q-values can take an arbitrary range, which is determined by the environment's reward function and the discount factor γ, which controls how strongly expected future rewards influence the agent's decisions and is in general specified by the environment definition. Depending on γ, the optimal Q-values for the same environment can take highly varying values, and can therefore be viewed as different learning environments themselves. In practice, it is not necessary that an agent learns the optimal Q-values exactly. As the next action at step t is chosen according to Equation (3.18), it is sufficient that the action with the highest expected reward has the highest Q-value for the sake of solving an environment presuming a deterministic policy. In other words, for solving an environment only the *descending order* of Q-values is important, and the task is to learn this correct order by observing rewards from the environment through interaction.

3.2.3 Policy gradients

As described above, when using a policy gradient method, the goal is to learn a parametrized policy directly based on a quantity $J(\boldsymbol{\theta})$, that in the fixed-horizon setting is equal to the value function (3.14),

$$J(\boldsymbol{\theta}) := V_{\pi_{\boldsymbol{\theta}}}(s). \tag{3.22}$$

In a gradient-based optimization procedure the parameters are updated according to

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla J(\boldsymbol{\theta}_t), \tag{3.23}$$

with a learning rate α, i.e., we perform gradient ascent on the parameters to maximize the expected return. The policy gradient theorem [109] then states that the gradient of the performance measure can be written as

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &= \nabla V_{\pi_{\boldsymbol{\theta}}}(s) \\ &\propto \sum_s \mu(s) \sum_a \nabla \pi_{\boldsymbol{\theta}}(a|s) Q_\pi(s, a) \\ &= \mathop{\mathbb{E}}_{\pi_{\boldsymbol{\theta}}} \left[\sum_a \nabla \pi_{\boldsymbol{\theta}}(a|S_t) Q_\pi(S_t, a) \right], \end{aligned} \tag{3.24}$$

where $\mu(s)$ is the on-policy distribution under the current policy, which depends on the time spent in each state, and S_t in the third line of Equation (3.24) are states sampled under the policy π. Using this, we can now derive the REINFORCE algorithm, that is the basis of policy gradient based training.

Our goal is to perform gradient ascent on the parametrized policy purely from samples generated from said policy through interactions with the environment. The last line of Equation (3.24) still contains a sum over all actions a, which we can replace by the sample $A_t \sim \pi$ after multiplying and dividing the terms in the sum by $\pi_{\boldsymbol{\theta}}(a|S_t)$,

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &\propto \mathop{\mathbb{E}}_{\pi_{\boldsymbol{\theta}}}\left[\sum_a \pi_{\boldsymbol{\theta}}(a|S_t) Q_\pi(S_t,a)\frac{\nabla\pi_{\boldsymbol{\theta}}(a|S_t)}{\pi_{\boldsymbol{\theta}}(a|S_t)}\right] \\ &= \mathop{\mathbb{E}}_{\pi_{\boldsymbol{\theta}}}\left[Q_\pi(S_t,A_t)\frac{\nabla\pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}}(A_t|S_t)}\right] \\ &= \mathop{\mathbb{E}}_{\pi_{\boldsymbol{\theta}}}\left[G_t\frac{\nabla\pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}}(A_t|S_t)}\right], \end{aligned} \tag{3.25}$$

where G_t is the expected return from Equation (3.13). Now, by using the fact that $\nabla \log x = \frac{\nabla x}{x}$, we can write

$$\mathop{\mathbb{E}}_{\pi_{\boldsymbol{\theta}}}\left[G_t\frac{\nabla\pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}}(A_t|S_t)}\right] = \mathop{\mathbb{E}}_{\pi_{\boldsymbol{\theta}}}\left[G_t \cdot \nabla \log \pi_{\boldsymbol{\theta}}(A_t|S_t)\right]. \tag{3.26}$$

This equation allows us to estimate the gradient of $J(\boldsymbol{\theta})$ simply by taking samples of interactions with the environment under the current policy $\pi_{\boldsymbol{\theta}}$, and leads us to the following parameter update in each iteration of the algorithm,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \, \nabla \log \pi_{\boldsymbol{\theta}}(A_t, S_t), \tag{3.27}$$

where α is again the learning rate, R_k is the reward, and T is the length of the episode. As mentioned above, value- and policy-based algorithms both have their strengths and shortcomings, and which of the two should be used depends on the environment. Indeed, both of these approaches, as well as their combination in the actor-critic framework, have been studied extensively classically as well as in a quantum setting, as we will see in the following section.

3.3 Quantum machine learning

After getting a basic understanding of quantum computing in Chapter 2, and the field of machine learning in this chapter, it is finally time to turn to the intersection

of these two fields: quantum machine learning (QML). As the focus of this thesis is on the near-term algorithms we described in Section 2.2.1, we will regard QML from the angle of VQAs, and will not discuss fault-tolerant algorithms, like the previously-mentioned one for solving linear systems of equations [5], in much detail. In Section 2.2.1, we have already seen how a gradient-based optimizer can be used to iteratively train the parameters of a quantum circuit, and how this can be applied to a number of different tasks in combinatorial optimization or quantum chemistry and simulation. When using these types of algorithms in a ML context, the parameter optimization scheme stays essentially the same. The only difference is that the parameters now have to be found for a set of training samples instead of just for the ground state of a single Hamiltonian, which introduces the same problems we have already studied for NNs above, like overfitting and the bias-variance tradeoff, to the realm of VQAs. Additionally, the quantum cousin of the vanishing gradient problem, the barren plateau phenomenon we have described in Section 2.2.1.2, remains to pose a challenge in the variational QML setting as well. Nonetheless, QML is a quickly evolving field [120, 121, 122], and these types of algorithms have been studied in a multitude of contexts, as we will see in the following section.

3.3.1 Near-term quantum machine learning

The QML algorithms that have been proposed for the NISQ-era can be broadly divided into two types: the VQAs we already know, which in this context are also referred to as quantum neural networks (QNNs)[1], and another approach that is based on the classical ideas of *kernel methods* [124]. Kernel methods are based on a similarity function called a *kernel* that maps data points into a *feature space*, and they make use of the so-called *kernel trick*. This trick allows computation of the similarity of data points in the potentially infinite-dimensional feature space given by the kernel, without ever having to compute the coordinates of the data points in that feature space directly. Instead, it suffices to only compute the inner products between the images of all pairs of data points to determine their similarity. A well-known example of a ML algorithm that utilizes kernels

[1]Note that there are also efforts to directly implement the classical neuron/perceptron model with a quantum approach [123], which is also sometimes referred to as a "quantum neural network." However, most of these are not suitable for NISQ devices so we do not consider them in this thesis. When we refer to QNNs, we mean parametrized quantum circuits that are trained with ML techniques to learn a certain task.

is the support vector machine (SVM) [125]. Simply put, the SVM is a linear classifier that learns to divide a set of points into two partitions, and does this by finding a separating hyperplane where the separation between data points in the two partitions is maximal. As this technique can only be used for linearly seperable data, it is combined with a kernel, where the kernel is chosen such that the given data points become linearly seperable in the high-dimensional feature space. The SVM then only has to effectively partition a linearly seperable data set. The idea of computing inner products of data points in a high-dimensional space suggests itself to be translated to the quantum domain, and indeed quantum kernel methods have enjoyed much popularity in the QML community in the past years [126, 26, 127, 128, 129]. One advantage of this type of model in the quantum setting is that the quantum device is only used to compute the kernel function, and therefore there is no need to variationally train circuit parameters. Furthermore, it has been shown that there exists a quantum kernel that allows for a rigorous separation between classical and quantum learners [128], even though it is for a contrived learning problem that does not directly translate to a task of practical relevance. Despite having no trainable parameters, however, it has been shown that quantum kernel methods suffer from a similar effect as the barren plateaus in VQAs, namely an exponential concentration of the inner products in the number of qubits [130].

In the classical literature, kernel methods were popular before the onset of deep learning that was a result of increased hardware performance in the early 2000's, and since then focus has substantially shifted to NNs in research as well as industry. Nonetheless, kernel methods are still interesting from the theoretical perspective, as they are much more amenable to obtaining rigorous results about their performance than NNs. Additionally, a certain type of kernel, called the *neural tangent kernel* (NTK), can be used to describe the evolution of certain NNs when they are trained with gradient descent [131]. With the NTK, theoretical results from kernel methods can be used to study NNs, and it was, for example, used to prove that wide enough NNs converge to a global minimum of the loss function [132, 133]. An analogous connection between QNNs and quantum kernel methods has been established as well [127, 134], and quantum versions of the NTK have also been studied [135, 136]. This connection between QNNs and kernel methods and its theoretical implications are a promising direction for future research.

QNNs themselves were studied in the context of a variety of different learning tasks in all the three main branches of ML. Early work in the field introduced QNNs for classification [137, 39, 138, 74], alongside classification with quantum kernel methods as mentioned above. Another well-studied area of QML is generative learning, a branch of unsupervised learning where the goal is to model a probability distribution that generated a given data set, to then produce similar samples that were not in the training set. As sampling from classically hard distributions is the basis of quantum supremacy experiments [2, 3], using PQCs to model probability distributions seems to be a promising direction of research. Quantum generative models can be broadly classified into three categories. The first are *quantum circuit born machines* (QCBMs) [139, 140, 28], that got their name from the fact that the model's probability to produce a certain sample is given by the Born rule. The second are quantum *generative adversarial networks* (GANs), inspired by classical GANs [141], where two models are trained together with opposing targets. The generator is trained to generate realistic samples of some data distribution, while the discriminator's goal is to detect the "fake" data generated by the generator. Several proposals for quantum GANs have been made [142, 143, 144, 145]. The third type of quantum generative model is related to classical Boltzmann machines [146], which are energy-based models that can be defined in terms of a Hamiltonian, and are closely related to Ising models and spin-glasses. Due to this connection to physics, Boltzmann machines naturally lend themselves to be adapted to the quantum domain, and indeed multiple proposals for variational quantum Boltzmann machines exist [147, 148], as well as general Hamiltonian-based models for learning thermal states [149].

Variational QML algorithms for the third branch of ML, reinforcement learning, are also an active area of research. Policy gradient methods, where the policy is parametrized in form of a PQC, have been explored in [150, 151, 76, 152], while Q-learning with PQCs as function approximators was studied in [153, 154, 75, 155, 156]. The combination of both types of models in the actor-critic framework was studied in [157, 158]. Variational RL is also the main type of learning explored in this thesis, where we study quantum Q-learning in Chapter 5, use it to train a model to solve instances of combinatorial optimization problems in Chapter 6, and evaluate the noise-robustness of variational proposals for Q-learning and policy gradients in Chapter 7, considering noise sources present on near-term quantum hardware.

Finally, it is also worth mentioning another type of variational QML that promises to be a fruitful direction for future research, namely hybrid quantum-classical algorithms. The term "hybrid" here does not refer to the fact that a classical model is used to find optimal parameters for a PQC, but is due to a classical and quantum model being trained together [159, 160, 161, 162, 163]. The idea behind this is that classical resources are leveraged as much as possible, and only those computations where a quantum computer is really beneficial are outsourced to the quantum model. As, especially for learning tasks that involve classical data, it can be expected that quantum resources will only be required for part of the overall computation, this approach enables getting the most out of the scarce quantum resources we expect to have in the near future. A key question in this setting is what exactly the classically hard parts in QML on a classical data set are or if they even exist in generic learning problems, for which so far, there is no clear answer, and this question is an interesting direction for future research.

3.3.2 Data encoding and the choice of ansatz

A major difference to VQAs used in the ML setting, as opposed to finding the ground states of Hamiltonians, is the fact that we now want to train a PQC given a set of training data samples. This raises the question of how to best provide the PQC with this data, which is especially important when we consider learning tasks on classical data, where this data has to be encoded into a quantum state that is accessible to the PQC.

One of the first ideas on how to efficiently store classical data in a quantum state was *amplitude encoding*, where a vector of n input values is stored in the amplitudes of a qubit register. This approach is efficient in the sense that it only requires logarithmically many qubits in the length of the input vector, so it is efficient in space. However, this comes at the cost of time, as in general, it takes time $\mathcal{O}(2^n)$ to prepare this state [164]. This means that amplitude encoding can require relatively deep circuits, so this approach is not practical for NISQ devices. Additionally, it turns out that classifying states resulting from amplitude encoded data can even be less efficient than classifying the original vectors directly [165], so care must be taken about the specific encoding that is used. When the data consists of bitstrings instead of real values, a simpler approach is to directly use these classical bitstrings as the inputs to a PQC by initializing each qubit in the register in the state $|0\rangle$ or $|1\rangle$, respectively, called *basis encoding*.

A third method, sometimes referred to as *qubit encoding* or *angle encoding* [166], inputs real-valued data as rotation angles of parametrized gates on the single- or multi-qubit level. While this method obviously does not share the advantage of being space-efficient, as in the case of amplitude encoding, qubit encoding can be performed with arbitrary parametrizable gates and shallow circuits, and it is therefore usually the number one choice in NISQ algorithms. However, care must be taken on how this encoding is performed, in order to ensure a high expressivity of the resulting QML model [167, 168, 169]. The authors of [167, 169] establish a connection between PQCs and Fourier series, where each PQC with input data x can be written as a truncated Fourier series $f(x)$ in the following way,

$$f(x) = \sum_{\omega \in \Omega} c_\omega e^{i\omega x}, \tag{3.28}$$

where Ω is the frequency spectrum of the Fourier series, which completely depends on the choice of gates used to encode the data. Therefore, the data encoding used in a QML model determines the complexity of the functions that this model can fit. If the encoding is too simple, for example when only a layer of parametrized X-rotations is used where the data is only encoded once, the model can only learn a sine function of its input, irrespective of the complexity of the rest of the circuit. One method to address this issue while keeping the encoding gates simple is to repeatedly encode the data in subsequent layers of the circuit (or equivalently in parallel on the level of the qubits), interspersed with the trainable parametrized gates [167, 168, 169]. This approach has been introduced under the name of *data re-uploading* in [168]. The number of repetitions of the data encoding directly influences the frequency spectrum of the Fourier series and, therefore, represents a hyperparameter that can be used to tune the model complexity in a PQC. While qubit encoding represents an easy and theoretically well-motivated method to encode data into a PQC, this direct correspondence between the dimension of the input vector and the number of qubits is somewhat problematic. Considering that state-of-the-art classical NNs can be trained on high-dimensional inputs, like thousands of pixels of an image, the question arises how these types of inputs can be efficiently encoded into PQCs without resorting to amplitude encoding. So far, there is no clear answer to this, and finding good data encoding techniques is an active area of research.

Along with the data encoding technique, a second architectural choice has to be made in designing a PQC for QML: the overall structure of the circuit. The

question of suitable ansatzes for QML is one that has only recently begun to be studied in the community. In lieu of ansatzes targeted for specific problems, an ansatz dubbed *hardware-efficient* (HWE) ansatz, that was originally introduced in [170], is often used. The idea behind this ansatz, as the name suggests, is to pick the circuit structure and gates in it according to the device it is targeted for, best utilizing the native connectivity of the quantum processing unit. Apart from this idea there is no clear definition of what a HWE ansatz looks like, but usually it consists of layers of parametrized single-qubit gates and additional entangling gates following a simple topology like a ring. While this ansatz can be a good choice for small-scale models, it is prone to suffer from barren plateaus (see Section 2.2.1.2) as the system size is scaled up. Therefore, it is clear that devising ansatzes that can be scaled up to a large number of qubits is of key importance.

One possibility to do this is to create problem-tailored ansatzes, and the nascent field of *quantum geometric learning*, inspired by the classical field of geometric deep learning introduced in Section 3.1.3, is a promising approach for doing this. The first ansatz of this type is called the quantum convolutional NN [55], in reference to classical convolutional NNs. The number of parameters in this ansatz grows only logarithmically with the input size, which makes it suitable for NISQ devices. In addition, it was proven that this ansatz does not suffer from the barren plateau phenomenon due to the shallowness of the resulting circuit [56]. A recent line of research also focuses on how to create circuits that are invariant or equivariant under certain group actions [171, 172, 173, 174, 175, 176, 57]. The permutation equivariant circuit that we introduce in Chapter 6 also belongs to this class of ansatzes. While these ansatzes are already promising, the search for scalable and performant problem-tailored quantum circuits has only just begun.

3.3.3 Is there potential for quantum advantage?

Possibly the most important question for near-term QML is whether we can expect any form of quantum advantage. As we have mentioned before, showing rigorous quantum advantage for VQAs is hard due to their heuristic nature, and empirical advantages are for now out of reach as the hardware is not there yet. In addition, quantum advantage, as opposed to quantum supremacy [2, 3], is not clearly defined so it is hard to give a definite answer to the above question. At least at the time of writing this thesis, there is no indication that variational QML will yield a speed-up on real-world, classical learning tasks.

Nonetheless, there are a number of interesting theoretical results that motivate further research in the field of QML. A prominent example is learning from *quantum data*, where data is fed into the QML model directly from a quantum source, i.e., an experiment or another quantum device. The idea behind this is similar to Richard Feynman's original quote that the most suitable tool to simulate a quantum system, is another quantum system. In the same vein, if you want to learn about quantum data, you better make the learning model quantum, too. It was shown that in this setting, there can be an exponential separation between classical and quantum learners [177]. Furthermore, the authors of [178] show that for some classically hard-to-compute functions, their outcomes can still be efficiently predicted by a classical ML model when it is given access to data generated by these functions.

In addition to speed-ups, the differences between classical and quantum NNs have also been studied from the angle of expressivity and the families of functions they can model. The authors of [179] investigate the expressivity of both types of models from an information geometric perspective, and introduce the effective dimension as a means to measure model complexity. They show that quantum models can have a higher effective dimension and a more favorable Fisher information spectrum compared to their classical counterparts. In a similar vein, but more specific to the generative learning setting, the authors of [28] show that there exist probability distributions that can be represented by a quantum circuit, but are hard to learn classically, and similar results were established in the context of the QAOA as well [180].

Looking beyond results that are deemed accessible to NISQ devices, there have been a number of works pertaining the families of functions that can be learned by quantum models. The authors of [128] establish en exponential separation between classical and quantum learners in the context on an SVM, where the data set is based on the discrete logarithm problem. In this setting, a quantum learner can efficiently compute the discrete logarithms of a given set of data points, and thereby turn a data set that looks essentially randomly labeled to a classical learner, into one that is linearly separable. This result has been extended to a RL setting for policy gradients in [150], showing the same exponential separation between classical and quantum RL agents, and we show in Section 5.2 that this separation also holds for certain types of environments in the Q-learning setting. Based on similar arguments from cryptography, the authors of [181] show an exponential separation for learning discrete distributions.

The above results show that there definitely exist functions that can not efficiently be learned with classical models. However, the settings in which this was shown are contrived and in a way reverse engineered by defining classically hard data sets on known classically hard tasks. The question remains whether these differences in quantum and classical models will eventually be beneficial on tasks that are practically relevant.