Qiskit-Torch-Module: Fast Prototyping of Quantum Neural Networks

Nico Meyer*†, Christian Ufrecht*, Maniraman Periyasamy*, Axel Plinge*, Christopher Mutschler*,
Daniel D. Scherer*, and Andreas Maier†
*Fraunhofer IIS, Fraunhofer Institute for Integrated Circuits IIS, Nürnberg, Germany
†Pattern Recognition Lab, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany

Abstract—**Quantum computer simulation software is an integral tool for the research efforts in the quantum computing community. An important aspect is the efficiency of respective frameworks, especially for training variational quantum algorithms. Focusing on the widely used `Qiskit` software environment, we develop the `qiskit-torch-module`. It improves runtime performance by two orders of magnitude over comparable libraries, while facilitating low-overhead integration with existing codebases. Moreover, the framework provides advanced tools for integrating quantum neural networks with `PyTorch`. The pipeline is tailored for single-machine compute systems, which constitute a widely employed setup in day-to-day research efforts.**

Index Terms—**quantum computing, variational quantum circuits, quantum machine learning, quantum simulation software**

I. Introduction

Frameworks for simulating quantum circuits have been the backbone of quantum computing research in the last decade. The collection of these tools is extensive, with the most widespread ones including `Qiskit` [1] and `Cirq` [2]. Depending on system constraints, it is possible to simulate between 15 to 25 noise-free qubits on common consumer hardware. With access to HPC compute qubit counts in the mid 40's are tractable [3], [4]. If one assumes additional constraints on the quantum system, approximate tensor network methods can be used to drastically increase this number [5]. Still, for many experiments a full simulation of the state is required. Especially for prototyping new algorithms and ideas most researchers do not have access to extensive hardware clusters.

One of the most promising candidates for making use of noisy intermediate-scale quantum (NISQ) devices are variational quantum algorithms (VQAs) [6], [7]. This paradigm is used in quantum machine learning (QML) [8], [9], where quantum neural networks (QNNs) play a central role. Other applications include quantum-enhanced optimization with QAOA [10], and reducing the circuit depth for chemical simulations [11] with quantum computers. However, variational approaches introduce the overhead of training the underlying variational quantum circuits (VQCs), which usually is done with gradient-based optimization. To keep developing these algorithms tractable, an efficient simulation framework for training the underlying quantum models is essential.

Related Work. The most widely used (counting GitHub interactions) quantum computing framework `Qiskit` offers its own toolbox for QML: `qiskit-machine-learning` [1]. Due to the prominence in the field, it is used by big parts of the quantum computing (QC) community to develop, test, and benchmark VQAs. This makes performance improvements of the framework highly relevant, both for the ease of prototyping new concepts, but also with regards to e.g. energy consumption associated with the training routine. Additional frameworks for training VQAs include `PennyLane` [12] and `TensorFlow Quantum` [13]. A recent study [14] suggests, that these toolboxes might provide performance improvements over `Qiskit`, and potentially also over `qiskit-machine-learning`. However, due to the different syntax and programming logic, re-factoring existing code between frameworks is a laborious task.

Contribution. We identify several shortcomings of `qiskit-machine-learning`, that limit the training efficiency of VQAs. Our proposed alternative resolves these issues and significantly reduces runtime overhead by about two orders of magnitude. Since many QML routines employ QNNs, we put special attention to this setup. While `qiskit-machine-learning` allows for a basic integration of QNNs with `PyTorch` [15], we streamline this connection and enable a more advanced usage. As both, `Qiskit` and `PyTorch`, are central to our framework, we name it `qiskit-torch-module` – short `qtm`. We emphasize that `qtm` is not intended to compete with general quantum simulation libraries. Instead, it should be viewed as a tool to speed up training VQAs compared to `qiskit-machine-learning`. With negligible code migration overhead we observe a reduction in end-to-end computation times from hours to minutes on a representative selection of QML algorithms. The `qtm` framework targets research efforts without access to extensive compute, where prototyping is limited to single-CPU desktop machines.

The paper is structured as follows: In Sec. II, we highlight the most important attributes of the `qiskit-torch-module`, namely simultaneous evaluation of observables, batch-parallelization, and an advanced integration with `PyTorch`. In Sec. III we benchmark the framework with respect to isolated metrics and on several end-to-end tasks. Finally, Sec. IV discusses the role of our work in the wider context of quantum computing simulators.

This research was conducted within the Bench-QC project, a lighthouse project of the Munich Quantum Valley initiative, which is supported by the Bavarian state with funds from the Hightech Agenda Bayern Plus.
Correspondence to: nico.meyer@iis.fraunhofer.de

II. Outline of the Module

The proposed qiskit-torch-module – from here on referred to as qtm – contains several sophisticated concepts that allow for a boost of performance and usability. As indicated in the introduction, the tool should be understood as an alternative to qiskit-machine-learning – abbreviated as qml – for the training of VQAs. While we have implemented several smaller alterations compared to qml, the main improvements constitute an efficient evaluation of multiple observables (Sec. II-A), batch parallelization (Sec. II-B), and a straightforward integration with PyTorch (Sec. II-C). Additionally, a small example of code migration is provided in Sec. II-D. For further information, the reader is encouraged to refer to the documentation of the qtm framework.

The common objective of the qml and our qtm framework is the training of VQAs, which incorporates quantum circuits with trainable parameters. A simple training objective can be interpreted as the cost function of a machine learning model with only one output

$$\mathcal{M}_{\Theta,\boldsymbol{s}}^{\text{simple}} = \langle 0 | U_{\Theta,\boldsymbol{s}}^{\dagger} O U_{\Theta,\boldsymbol{s}} | 0 \rangle \tag{1}$$

$$=: \langle O \rangle_{\Theta,\boldsymbol{s}} \,, \tag{2}$$

with some arbitrary parameterized ansatz $U_{\Theta,\boldsymbol{s}}$. Here, Θ denotes a set of trainable parameters, $\boldsymbol{s}$ refers to data encoding, and O is an observable.

A. Efficient Evaluation of Multiple Observables

In practice, the envisioned quantum model is often more general than described in Eqs. (1) and (2). Typically, multiple observables are measured, which allows for an adjustable output size of the model. The generalized definition for M observables reads:

$$\mathcal{M}_{\Theta,\boldsymbol{s}} = \begin{bmatrix} \langle O_0 \rangle_{\Theta,\boldsymbol{s}} \\ \vdots \\ \langle O_{M-1} \rangle_{\Theta,\boldsymbol{s}} \end{bmatrix} \tag{3}$$

If executed on actual quantum hardware, only jointly measurable observables can be evaluated in parallel. However, for simulation, this restriction does not apply. Explicitly evolving

$$|\psi_{\Theta,\boldsymbol{s}}\rangle := U_{\Theta,\boldsymbol{s}} \, |0\rangle \tag{4}$$

only once allows to determine all expectation values via post-processing following $\langle O_i \rangle_{\Theta,\boldsymbol{s}} = \langle \psi_{\Theta,\boldsymbol{s}} | O_i | \, \psi_{\Theta,\boldsymbol{s}} \rangle$. Exploiting this simplification, the qtm framework achieves an approximately M-fold speed-up compared to qml, which evolves the state for each observable.

For training the model it is typically necessary to compute gradients w.r.t. the (trainable) parameters, i.e.

$$\nabla_{\Theta} \mathcal{M}_{\Theta,\boldsymbol{s}} = \begin{bmatrix} \nabla_{\Theta} \langle O_0 \rangle_{\Theta,\boldsymbol{s}} \\ \vdots \\ \nabla_{\Theta} \langle O_{M-1} \rangle_{\Theta,\boldsymbol{s}} \end{bmatrix}. \tag{5}$$

A frequently employed method to compute these gradients is the parameter-shift rule [16], which is also compatible with estimation on actual hardware. Unfortunately, this approach requires the simulation of $2 \cdot |\Theta|$ circuits – independent of the number of observables, if only evolving each state once as described before. This leads to long training times, even for small systems. Alternatively, it is possible to acquire an SPSA-approximation of the gradients with only 2 circuit evaluations – also independent of M. However, this technique often leads to a less stable and longer training routine [17]. If restricting to simulation, the *reverse* gradient estimation technique – also referred to as *adjoint method* [18] – is a promising tool for small to medium-scale systems. It re-uses information from the forward pass and executes the quantum circuit in reverse, by applying adjoint operation to the intermediate states. The originally proposed method, which is also implemented in qml, is defined only for single-observable quantum models. To allow for efficient computation of gradients for multiple observables, we extend the original algorithm (i.e. 'Algorithm 1' in [18]). This is done in a similar manner as previously described for the expectation values. As the approach contains interleaved state evolution and evaluation of observables, the performance gain tends to be smaller compared to evaluating expectation values. However, as discussed in Sec. III, the improvements over qml are still considerable.

When using the qtm framework with multiple observables, these techniques are employed by default. The underlying routine for estimating expectation values is based on qiskit.primitives.Estimator, the gradients are computed with a modified version of qiskit_algorithms.ReverseEstimatorGradient.

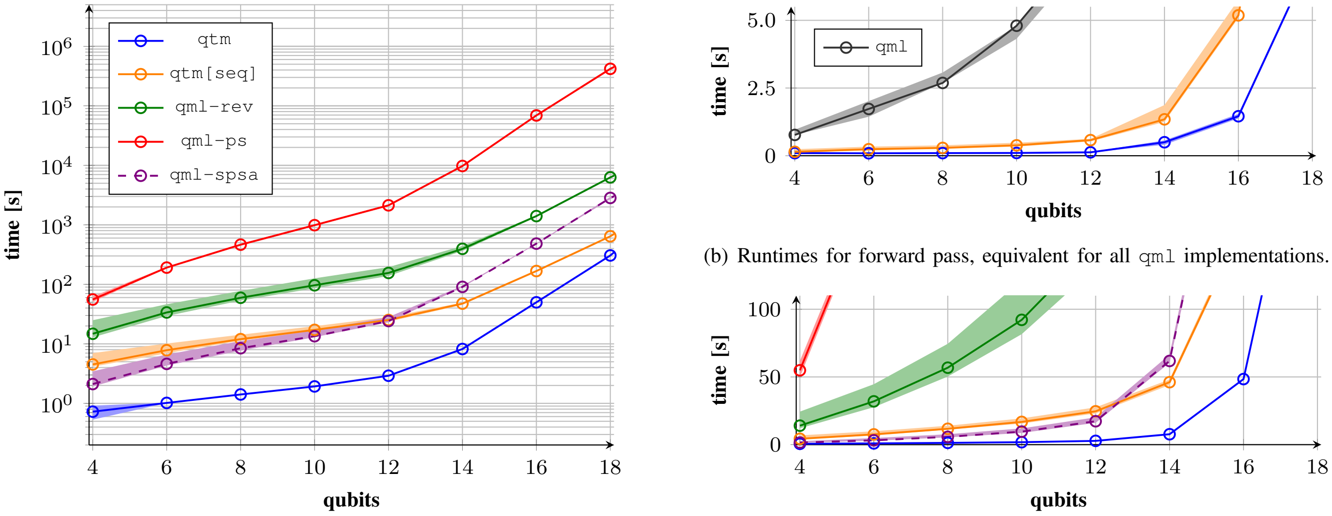
B. Parallelization for Batched Inputs

In a typical (quantum) machine learning setting, the model is trained in a batched manner. This requires the computation of expectation values and gradients for multiple inputs $\boldsymbol{s}^{(i)},\ i = 0, \dots, B-1$ and a fixed set of parameter values Θ:

$$\mathcal{M}_{\Theta,\left[\boldsymbol{s}^{(i)}\right]_{i=0,\dots,B-1}} = \begin{bmatrix} \mathcal{M}_{\Theta,\boldsymbol{s}^{(0)}} \\ \vdots \\ \mathcal{M}_{\Theta,\boldsymbol{s}^{(B-1)}} \end{bmatrix} \tag{6}$$

An equation for the gradients w.r.t. the parameters can be defined in complete correspondence to Eq. (6). As all B experiments are independent, it is possible to distribute the workload among multiple threads. If the batch size is divisible by the number of threads T, we equally distribute the task of handling the inputs $\boldsymbol{s}^{(i)}$ among all workers. Otherwise, we assign the first $B - \left(T \cdot \lfloor \frac{B}{T} \rfloor\right)$ threads to an additional task, to keep workloads as balanced as possible. The results of all workers are collected after completion, resulting in an array of shape (B, M) for expectation values and $(B, M, |\Theta|)$ for gradients.

As will be demonstrated in Sec. III, this batch-wise parallelization technique is much more efficient than either the shot-based or circuit-based parallelization approach implemented in qml. Depending on the actual device, for system sizes of up to 14 qubits this gives a close to T-fold speed-up. For larger systems the improvement is limited by the increasing resource requirements for state evolution, but is still noticeable.



(a) Combined computation times for forward and backward pass.

(b) Runtimes for forward pass, equivalent for all qml implementations.

(c) Runtimes for backward pass, qml-spsa only gives approximation.

Fig. 1: Benchmarking results of the proposed qtm module, compared to the available implementations in qml. All experiments are averaged over 10 independent runs with standard deviations depicted in pale colors. The runtimes refer to a batch size of $B = 48$, with single-qubit Pauli-Z observables on all qubits, and a circuit depth of $d = 3$. The number of trainable parameters scales linearly in the number of qubits. (a) Depicts the combined times for one pass of computing expectation values and gradients on a logarithmic scale; (b) depicts the times for the forward pass, i.e. computing expectation values, with a cut-off at 5 seconds; (c) depicts the times for the backward pass, i.e. computing gradients, with a cut-off at 100 seconds.

C. Automatic Differentiation and Hybrid Models

An important aspect of any software framework is its usability. For machine learning this entails automatic tracking of gradients, as realized e.g. by the autograd functionality of PyTorch. In fact, also the qml module provides such an interface via the TorchConnector. However, we introduce extended functionalities that are highly desirable in practice.

Many VQAs use an underlying QNN with multiple sets of trainable parameters, e.g. the *standard* variational parameters $\boldsymbol{\theta}$ and additional input scaling parameters $\boldsymbol{\lambda}$. As both play different roles in the underlying model, typically also the optimal hyperparameters – e.g. learning rates or initialization strategies – vary. While qml only allows to subsume all trainable parameters in a single set with fixed hyperparameters, qtm allows for a fine-grained setup. We demonstrate in Sec. III-B that this flexibility can indeed make a substantial difference in practice. For parameter initialization, our implementation prevents unexpected behavior caused by Qiskit's value-to-parameter assignment strategy. It is always done in alphabetical order, which requires the user to guarantee consistency of Parameter naming with the order of provided values. The qtm module takes care of this pitfall and enforces the more intuitive ordering defined during model construction.

We also provide a HybridModule, which is end-to-end differentiable and consists of three parts: (i) a classical fully connected pre-processing layer with user-defined input size, with the output encoded in the consecutive QNN; (ii) an instance of a QuantumModule with single-qubit observable measurements; (iii) a classical fully connected post-processing layer, receiving the output from the QNN, and user-defined output size; this allows for an abstraction of problem dimensionality and qubit count, as often done in the literature.

D. Migration Guide from qml to qtm

It is straightforward to migrate implementations using qml to qtm. Let us assume an QNN-based approach, with VQC denoting the underlying quantum circuit consisting of data encoding FEATURE_MAP and variational layers ANSATZ:

```
import qiskit_machine_learning as qml

# step 1: set up quantum neural network
qnn = qml.EstimatorQNN(circuit=VQC,
          input_params=FEATURE_MAP.parameters,
          weight_params=ANSATZ.parameters)
# step 2: connect to PyTorch, init to [0, 2*pi]
p = random.uniform(0, 2*pi, size=qnn.num_weights)
model = qml.TorchConnector(qnn, initial_weights=p)
```

As demonstrated, qml requires to first explicitly define a QNN, which subsequently is connected to Pytorch. In qtm, the code snippet would be simplified as follows:

```
import qiskit_torch_module as qtm

# set up PyTorch module and init to [0, 2*pi]
model = qtm.QuantumModule(circuit=VQC,
            encoding_params=FEATURE_MAP.parameters,
            variational_params=ANSATZ.parameters,
            variational_params_initial="uniform")
```

The model can subsequently be used for training and inference as any arbitrary PyTorch module. In more general VQAs, the sub-modules FastEstimator and FastReverseEstimatorGradient can be used as standalone tools. As the modifications described in Secs. II-A and II-B are realized on this level, significant speed-ups over qiskit's native functionalities can be expected. The QuantumModule used above just provides an additional level of abstraction for convenient usage.

III. Benchmarking Results

To quantify the benefits of qiskit-torch-module over qiskit-machine-learning, we benchmark raw performance of computing expectation values and gradients in Sec. III-A, backed up with end-to-end examinations in Sec. III-B. We denote the proposed module as qtm, with qtm[seq] representing a sequential version without the techniques described in Sec. II-B. We compare performances to Qiskit's qml, with qml-rev using reverse gradient computation, qml-ps resorting to the parameter-shift rule, and qml-spsa approximating gradients with SPSA.

We select an ansatz that is frequently employed in QML, using data re-uploading [19] and trainable input scaling [20]. For n qubits this can be expressed as

$$U_{\Theta,\boldsymbol{s}} = U_{\boldsymbol{\theta}_d,\boldsymbol{\lambda}_d,\boldsymbol{s}} \dots U_{\boldsymbol{\theta}_1,\boldsymbol{\lambda}_1,\boldsymbol{s}}, \tag{7}$$

where d denotes the depth and $\boldsymbol{s}$ is a fixed data vector. The set $\Theta := (\boldsymbol{\theta}, \boldsymbol{\lambda})$ entails variational parameters $\boldsymbol{\theta}$ and state-scaling parameters $\boldsymbol{\lambda}$. The individual unitaries are realized with parameterized single-qubit rotations and a nearest-neighbor structure of CNOT-gates. The number of trainable parameters scales linearly in both the number of qubits and the circuit depth.

All experiments were conducted using qiskit v1.0, qiskit-algorithms v0.3.0, torch v2.2.1, qml v0.7.1, and qtm v1.0. While we recommend using the module with qiskit's stable release, we also ensured backwards compatibility down to qiskit v0.44. All experiments in Secs. III-A and III-B were performed on a system running Ubuntu 23.10, with 32 GB of RAM and a AMD Ryzen 9 5900X 12-core CPU. In Sec. III-C we perform an ablation study with additional hardware arrangements and operating systems. More details are also provided in the README of the enclosed framework.

A. Computing Expectation Values and Gradients

The runtime needed for training a VQA is dominated by the resources required for computing expectation values and gradients. Three features of interest are the runtime scaling with the system size, with the depth of the underlying VQC, and the number of observables.

In Fig. 1, we show the results of a runtime analysis for varying numbers of qubits. The trainable parameter count scales linearly in the system size. As expected, the resource requirements increase exponentially for all frameworks. However, qtm provides a speed-up of close to two orders of magnitude over qml-rev, and of one order of magnitude over qml-spsa, which only computes approximate gradients. For systems sizes over 14 qubits the efficiency of batch-parallelization decreases slightly, since evolving the exponentially large state vector starts to dominate the total runtime. Still, even the sequential version qtm[seq] demonstrates a clear improvement. An additional runtime analysis of forward and backward pass reveals that the overall resources are largely consumed by gradient computation. We demonstrate in Sec. III-B that the depicted absolute improvements transfer to end-to-end settings and therefore can help reduce prototyping and development overhead.

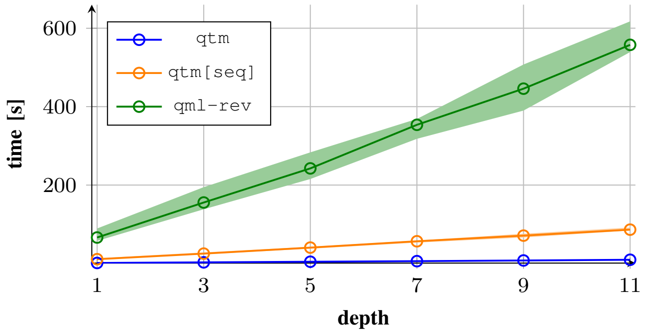


Fig. 2: Benchmarking results of the proposed qtm module, compared to qml-rev. The values depict gradient computation times for a batch size of $B = 48$, with single-qubit Pauli-Z observables on all 12 qubits. The number of trainable parameters scales linearly in the depth. All experiments are averaged over 10 independent runs with standard deviations depicted in pale colors. The times for qml-ps are too long to appear in the plot, qml-spsa was intentionally omitted since it only calculates approximate gradients.

Factor	observables							
Forward	1	2	4	8	12	16	24	32
qtm	5.60	11.4	23.2	44.4	66.6	81.5	131	172
qtm[seq]	1.17	2.15	4.67	9.33	14.0	18.7	25.9	34.5
Factor	**observables**							
Backward	1	2	4	8	12	16	24	32
qtm	8.35	15.6	27.4	43.5	54.0	60.1	68.3	74.3
qtm[seq]	1.01	1.82	3.11	4.79	5.91	6.61	7.47	8.08

Tab. I: Improvement factors over qml-rev implementation for varying numbers fo measured observables. The values are averaged over 10 independent runs of batch size 48 and refer to 12-qubit circuits with depth 3, i.e. 168 trainable parameters.

The scaling with the circuit depth – typically proportional to the number of trainable parameters – is shown in Fig. 2. The runtime scales linearly with the number of trainable parameters which is again proportional to the circuit depths. In agreement with the previous evaluations, the runtime is clearly reduced when using qtm. Especially when training models with increasing expressivity (related to parameter count), we expect a huge practical difference.

We compare the performance of both versions of qtm to the best-performing implementation qml-rev for a increasing number of observables in Tab. I. This is highly relevant for e.g. multi-class classification tasks, or reinforcement learning (RL) environments with large action spaces. Additionally, it is also of interest for hybrid modules, where the output of a QNN is fed into a classical neural network for post-processing. The results demonstrate the superiority of qtm in all scenarios. Indeed, for the forward pass, already qml[seq] improves upon qml-rev by a factor corresponding to the number of observables. For gradient computation, this improvement is slightly less significant, due to the complex nature of the reverse gradient estimation routine (for details see Sec. II-A). Still, especially with activated batch-parallelization, we report improvements on the order of one to two magnitudes.

B. Comparing End-to-End Performance

While the isolated benchmarks in Sec. III-A are helpful to anticipate the performance of the frameworks in a general setting, most important is the relevance for actual research work. In the following, we compare existing implementations using `qml` to a migrated version based on `qtm`. We want to emphasize, that the required re-factoring is typically very limited – see also the examples provided with the framework. We selected the examples from two different sub-fields of QNN-based QML, namely quantum-enhanced classification and quantum reinforcement learning. We are confident that these benchmarks demonstrate the relevance of the proposed framework for a broader scope.

On of the most common applications of QNNs is quantum-enhanced classification. Given some input data, the model is trained to predict the associated ground-truth label. A widely used benchmark dataset is MNIST [21], consisting of handwritten digits with class labels from from 0 to 9. We select a full-quantum approach – i.e. without trainable classical parameters. The image is encoded on 10 qubits using incremental data-uploading [22], interleaved with overall 220 trainable parameters. The expectation value of single-qubit Pauli-Z observables is measured and post-processed using a softmax function. We migrate the original `qml`-based implementation to the `qtm` library – with adjustments only being required for the setup of the quantum model.

Training is performed for 250 epochs with a batch size of 48 for each parameter update and a learning rate of 0.001. The results from both implementations are in agreement with those of the original paper [22], where the quantum model achieved an accuracy of about 55% after training. Note that a random classifier would only correctly label one out of ten samples. As we are especially interested in the runtime, we compare the two realizations in Tab. II. It is evident, that using `qtm` instead of `qml` drastically reduces the time required for training the quantum classifier. The end-to-end execution time is reduced from about 14 hours to 14 minutes, constituting a 60-fold improvement. This agrees with the results from Sec. III-A, where the improvement for a 10-qubit system with 10 observables was predicted to be in the 1.5 to 2 orders of magnitude range. This reduction in overall computation time enables developing refinements of the examined algorithm, as modifications can now be evaluated in a reasonable timeframe. This might allow to e.g. identify more sophisticated circuit ansätze that guarantee an accuracy closer to the classically achievable 98%.

	total time	total steps	time per step
`qtm`	838s	12000	69.8ms
`qml-rev`	50804s	12000	4233.7ms

Tab. II: Statistics for training the full-quantum classification algorithm [22] on the MNIST dataset. The 10-qubit VQC incorporates 220 trainable parameters. Pauli-Z expectation values are measured on all 10 qubits and undergo softmax post-processing before label prediciton. Training is performed for 250 epochs with a batch size of 48 samples each.

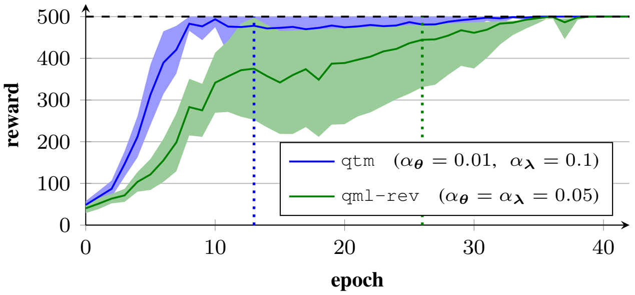


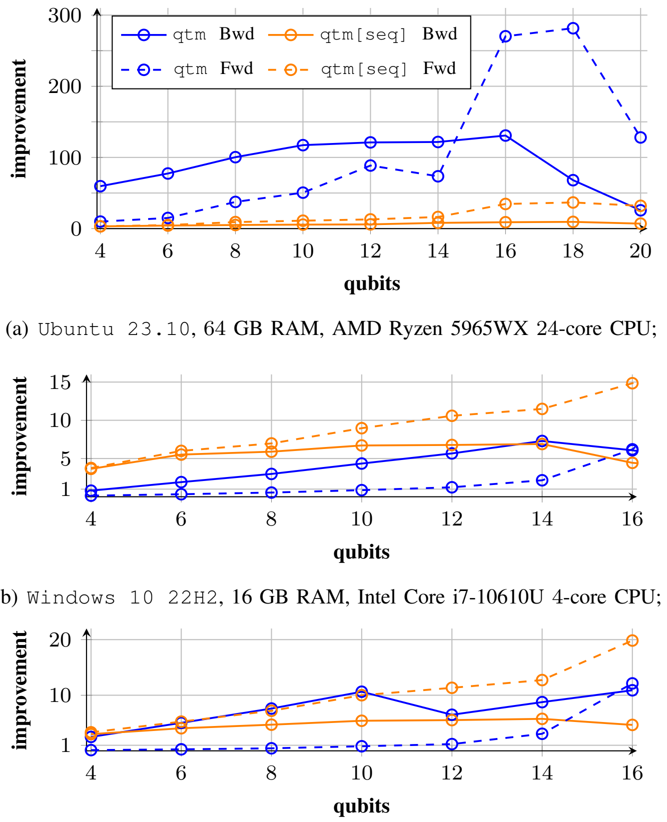
Fig. 3: RL performance of the QPG algorithm proposed in [23] on `CartPole-v1` for 100 random initializations. The performance difference originates from individual learning rates that can be set in `qtm`. In contrast, `qml` does not provide this option. Apart from this, also the runtime for each update step is reduced as depicted in Tab. III.

	total time	total steps	time per step
`qtm`	145.7s	37080	3.93ms
`qml-rev`	2301.1s	71259	32.29ms

Tab. III: Statistics for training the QPG algorithm [23] on the `CartPole-v1` environment, averaged over 10 independent runs. The `qtm` module overall requires less steps to achieve the same performance due to flexibility of learning rates. Additionally, `qtm` also demonstrates an about 10-fold reduction of runtime for each training step.

Quantum reinforcement learning (QRL) has emerged as a prominent area within the field of QML [9], with QNNs being used as function approximators. We select a quantum policy gradient (QPG) approach [20], which is based on direct parameterization of the policy, i.e. the intended behavior. We build upon an implementation of a refined version of QPG [23], which employs the QML framework. This setup allows to highlight another advantage of `qtm`, namely the possibility to use different hyperparameters for multiple parameter sets. We use the ansatz employed in the original work [23], which incorporates trainable variational parameters $\boldsymbol{\theta}$ and state scaling parameters $\boldsymbol{\lambda}$. In the original implementation that utilizes the `qml` module, it is necessary to apply identical learning rates for both parameter sets.

The `CartPole-v0` experiment executed on 4 qubits, depicted in Fig. 3, highlights the limitations of this rigidity. A grid search for learning rates ranging from 0.01 to 0.1 identified the optimal unified learning rate to be 0.05 for both $\alpha_{\boldsymbol{\theta}}$ and $\alpha_{\boldsymbol{\lambda}}$. However, employing the `qtm` module allows to identify the distinct learning rates $\alpha_{\boldsymbol{\theta}} = 0.01$ and $\alpha_{\boldsymbol{\lambda}} = 0.1$. This results in significantly faster and smoother convergence. The benefits are evident in the overall training duration as presented in Tab. III, where the time per experiment was dramatically reduced from approximately 40 minutes to merely 2.5 minutes. While the about 10-fold runtime reduction for each training step is still significant, the improvement is smaller compared to the classification task. This can be attributed to the system size of only 4 qubits and the evaluation of only a single observable. Nonetheless, both the overall computation time and also the flexibility of defining multiple hyperparameter sets highlights the benefits of `qml` over `qml`.



(a) `Ubuntu 23.10`, 64 GB RAM, AMD Ryzen 5965WX 24-core CPU;

(b) `Windows 10 22H2`, 16 GB RAM, Intel Core i7-10610U 4-core CPU;

(c) `macOS Ventura 13.6.1`, 32 GB RAM, Intel Core i7 6-core CPU;

Fig. 4: Ablation study testing efficiency of `qtm` on different hardware configurations and operating systems. The plots denote the runtime improvement factor over `qml-rev` on the same hardware for forward (Fwd) and backward (Bwd) pass. To ensure comparability the batch size is always selected as 4 times the number of physical CPU cores of the respective setups.

C. Ablation Study of Hardware and Operating System

To demonstrate the efficiency of `qtm` for different hard- and software setups, we report the improvement factor compared to running the same experiment using `qml` in Fig. 4.

The most significant speed-up is achieved on `Linux`-based operating systems as apparent from Fig. 4(a), where up to 300-fold runtime improvements are demonstrated. This is consistent with the results reported in previous sections where the optimal performance gain is given by the number of observables M times the available threads T. Indeed, the forward pass for 16 qubits comes close to $24 \cdot 16 = 384$. The absolute combined computation time for one input batch is reduced from 49 minutes to 21 seconds for this configuration.

To evaluate cross-platform compatibility we benchmarked `qtm` on notebooks with `Windows` in Fig. 4(b) and `MacOS` in Fig. 4(c). The expected speed-up is limited by the inefficiency of Python's `multiprocessing` library on these operating systems. Overall, however, we still observe a performance improvement proportional to the number of measured observables M. These results underline the usefulness of our framework for various hard- and software setups.

IV. Discussion and Broader Scope

We introduced the `qiskit-torch-module`, a framework for training variational quantum algorithms (VQAs), with a special focus on quantum neural networks (QNNs). It is based on a integration of `Qiskit` [1] and `PyTorch` [15] and resembles the structure of the respective functionalities from `qiskit-machine-learning`. The main modifications described in Sec. II include improved measurement of multiple observables, batch parallelization, and extended functionalities for automated gradient computation. As demonstrated in Sec. III, this leads to improved performance and usability of the proposed framework. This enables prototyping quantum machine learning (QML) algorithms on broadly available hardware in merely minutes.

We acknowledge that there are quantum software libraries like `PennyLane` [12] or `TensorFlow Quantum` [24] that potentially deliver comparable performance [14]. However, within the realm of `Qiskit`-based implementations for training of variational quantum circuits (VQCs), our framework clearly improves upon the previous state of the art. It has to be noted that `qtm` was developed for classical simulation of quantum circuits [18]. Some algorithmic improvements could be transferred to actual hardware by incorporating other primitives such as the parameter shift rule and SPSA-based approximations [25]. Given the constraints of noisy intermediate-scale quantum (NISQ) devices and current access modalities, the classical simulation of VQAs will remain an important part of research. The modular structure of `qtm` allows for an easy extension, e.g. the integration of quantum natural gradients [26], which are frequently used in QML [27], [28].

Altogether, we envision `qtm` as a convenient replacement of `qml`, in the sense that almost no code re-factoring is required for existing implementations. Furthermore, the syntax closely resembles that of `qml` and therefore is easy to use for researchers already familiar with this framework. The performance improvements help to reduce prototyping times from several hours to merely minutes, drastically simplifying the task of developing and refining variational algorithms.

Acknowledgment

We acknowledge the use and modification of parts of `Qiskit`'s code-base as a backbone of our implementation. Furthermore, the framework partially emulates structural components of `qiskit-machine-learning`, with modifications highlighted in this paper. Further details are provided in the documentation and source code of the `qiskit-torch-module`.

Code Availability and Compatibility

The described library can be installed via `pip install qiskit-torch-module`. The code is also available at https://github.com/nicomeyer96/qiskit-torch-module. The default setup imports `qiskit v1.0` but compatibility down to `qiskit v0.44` is ensured. The enclosed `README` provides setup details and usage instructions. Further information and data is available upon reasonable request.

REFERENCES

[1] Qiskit contributors, "Qiskit: An open-source framework for quantum computing," 2023.

[2] Cirq developers, "Cirq: An open source framework for programming quantum computers," 2023.

[3] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong, "Full-state quantum circuit simulation by using data compression," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–24.

[4] H. Bayraktar, A. Charara, D. Clark, S. Cohen, T. Costa, Y.-L. L. Fang, Y. Gao, J. Guan, J. Gunnels, A. Haidar *et al.*, "cuquantum sdk: a high-performance library for accelerating quantum science," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 1, 2023, pp. 1050–1061.

[5] S. Patra, S. S. Jahromi, S. Singh, and R. Orus, "Efficient tensor network simulation of IBM's largest quantum processors," *arXiv:2309.15642*, 2023.

[6] J. Preskill, "Quantum computing in the nisq era and beyond," *Quantum*, vol. 2, p. 79, 2018.

[7] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio *et al.*, "Variational quantum algorithms," *Nat. Rev. Phys.*, vol. 3, no. 9, pp. 625–644, 2021.

[8] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, "Quantum machine learning," *Nature*, vol. 549, no. 7671, pp. 195–202, 2017.

[9] N. Meyer, C. Ufrecht, M. Periyasamy, D. D. Scherer, A. Plinge, and C. Mutschler, "A Survey on Quantum Reinforcement Learning," *arXiv:2211.03464*, 2022.

[10] N. Moll, P. Barkoutsos, L. S. Bishop, J. M. Chow, A. Cross, D. J. Egger, S. Filipp, A. Fuhrer, J. M. Gambetta, M. Ganzhorn *et al.*, "Quantum optimization using variational algorithms on near-term quantum devices," *Quantum Sci. Technol.*, vol. 3, no. 3, p. 030503, 2018.

[11] H. R. Grimsley, S. E. Economou, E. Barnes, and N. J. Mayhall, "An adaptive variational algorithm for exact molecular simulations on a quantum computer," *Nat. Commun.*, vol. 10, no. 1, p. 3007, 2019.

[12] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, S. Ahmed, V. Ajith, M. S. Alam, G. Alonso-Linaje, B. AkashNarayanan, A. Asadi *et al.*, "Pennylane: Automatic differentiation of hybrid quantum-classical computations," *arXiv:1811.04968*, 2022.

[13] M. Broughton, G. Verdon, T. McCourt, A. J. Martinez, J. H. Yoo, S. V. Isakov, P. Massey, R. Halavati, M. Y. Niu, A. Zlokapa *et al.*, "Tensorflow quantum: A software framework for quantum machine learning," *arXiv preprint arXiv:2003.02989*, 2021.

[14] A. Jamadagni, A. M. Läuchli, and C. Hempel, "Benchmarking quantum computer simulation software packages," *arXiv:2401.09076*, 2024.

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[16] G. E. Crooks, "Gradients of parameterized quantum gates using the parameter-shift rule and gate decomposition," *arXiv:1905.13311*, 2019.

[17] M. Wiedmann, M. Hölle, M. Periyasamy, N. Meyer, C. Ufrecht, D. D. Scherer, A. Plinge, and C. Mutschler, "An empirical comparison of optimizers for quantum machine learning with spsa-based gradients," in *IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 1, 2023, pp. 450–456.

[18] T. Jones and J. Gacon, "Efficient calculation of gradients in classical simulations of variational quantum algorithms," *arXiv:2009.02823*, 2020.

[19] A. Pérez-Salinas, A. Cervera-Lierta, E. Gil-Fuster, and J. I. Latorre, "Data re-uploading for a universal quantum classifier," *Quantum*, vol. 4, p. 226, 2020.

[20] S. Jerbi, C. Gyurik, S. Marshall, H. Briegel, and V. Dunjko, "Parametrized quantum policies for reinforcement learning," *Adv. Neural Inf. Process. Syst.*, vol. 34, pp. 28 362–28 375, 2021.

[21] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits," 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[22] M. Periyasamy, N. Meyer, C. Ufrecht, D. D. Scherer, A. Plinge, and C. Mutschler, "Incremental data-uploading for full-quantum classification," in *IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2022, pp. 31–37.

[23] N. Meyer, D. Scherer, A. Plinge, C. Mutschler, and M. Hartmann, "Quantum Policy Gradient Algorithm with Optimized Action Decoding," in *International Conference on Machine Learning (ICML)*, vol. 202. PMLR, 2023, pp. 24 592–24 613.

[24] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv:1606.01540*, 2016.

[25] M. Periyasamy, A. Plinge, C. Mutschler, D. D. Scherer, and W. Mauerer, "Guided-spsa: Simultaneous perturbation stochastic approximation assisted by the parameter shift rule," *arXiv:2404.15751*, 2024.

[26] J. Stokes, J. Izaac, N. Killoran, and G. Carleo, "Quantum Natural Gradient," *Quantum*, vol. 4, p. 269, 2020.

[27] N. Meyer, D. D. Scherer, A. Plinge, C. Mutschler, and M. J. Hartmann, "Quantum natural policy gradients: Towards sample-efficient reinforcement learning," in *IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 2, 2023, pp. 36–41.

[28] S. Thanasilp, S. Wang, N. A. Nghiem, P. Coles, and M. Cerezo, "Subtleties in the trainability of quantum machine learning models," *Quantum Mach. Intell.*, vol. 5, no. 1, p. 21, 2023.