QHACK 2021: QOSE
Quantum Optimal Subarchitecture Extration

Team: Many body system

Aroosa Ijaz, Jelena Mackeprang, Kathleen Hamilton, Roeland Wiesema, Yash Chitgopekar

February 26, 2021

Summary

We have adapted the Optimal Subarchitecture Extraction algorithm (OSE) recently proposed in [7] to find the optimal ansatz for a quantum circuit classifier. Our implementation is named QOSE (Quantum Optimal Subarchitecture Extraction, pronounced as "cosy/cozy") is written in PennyLane and uses a search tree to identify optimal ansatz based on: parameter size, cost to simulate, and error rate.

1 Power up pitch

Project Description:

Training a quantum classifier on a classical data set is no easy feat, and one of the key issues for getting good performance is choosing a suitable ansatz that fits the problem of interest well. Due to the absence of practical guidelines for circuit design, we aim to leverage the full software stack available to the QML practitioner to find the optimal circuit architecture by a tree-based metaheuristic. Inspired by an analogous approach in classical machine learning, we propose Quantum Optimal Subarchitecture Estimation, or QOSE.

By combining the the scalability of AWS with the versatility and performance of Xanadu's PennyLane, we can reliably train hundreds of different circuit architectures in parallel. In QOSE, we iteratively add layers to a simple base circuit, increasing the expressiveness as the circuit deepens. This construction process can be described by a directed graph in the form of a tree, where nodes correspond to circuits of depth d. For each node in the tree, we briefly train the corresponding circuit on a subset of the actual data to get a glimpse of the expected performance.

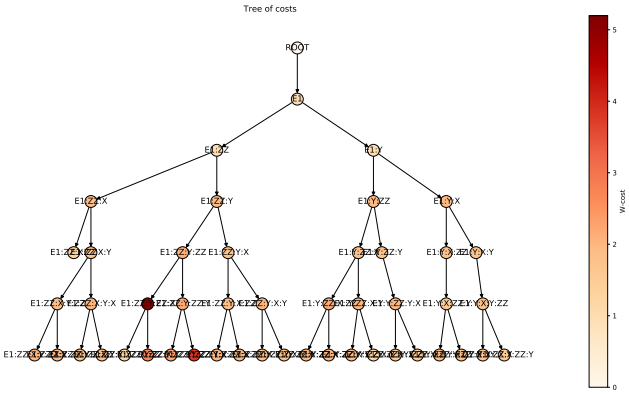

Figure 1: The search tree generated over a search space of 3 gate layers ZZ (control-Z), X (layer of X rotations), and Y (layer of Y rotations). Serial, local execution.

To combat the exponential increase of the search space of possible circuit architectures, we use a tree-pruning algorithm that eliminates nodes based on each circuit's quality. This quality is determined by a metric that combines inference speed, accuracy and model size into a single scalar cost. Our code can be made embarrassingly parallel by evaluating the cost of each depth-d node concurrently.

Source code

https://github.com/kmz4/QHACK2021/tree/main/src

Resource Estimate:

We plan to deploy our code on AWS in the upcoming days. The circuits we plan to run will be small, hence we will only require the local simulator keeping costs reasonable.

1. For this initial demonstration we limit our choice of classifier gates to the set: [ZZ,X,Y]. The possible embeddings will be [Angle Embedding, Amplitude Embedding, QAOA Embedding, XXZ Embedding, Aspuru Embedding, Random Embedding]

2. This means that at most (without pruning our search tree), we will need to calculate $n_{\text{embedding}} \times 3^d$ circuits. If for each circuit in the tree, we do a quick hyper-parameter search over optimal batch size and learning rate, this will add a factor of $n_{\text{batch sizes}} \times n_{\text{learning rates}}$ bringing the total number of circuits that have to be trained to $N = n_{\text{embedding}} \times 3^d \times n_{\text{batch sizes}} \times n_{\text{learning rates}}$.

3. Each circuit costs about 10-30 seconds to train, depending on the depth, hence for worst case we will need $N \times 30s$ of computing power.

4. We propose to use MPI4PY to handle the parallelization on AWS for calculating individual circuits. Due to the embarrassingly parallel nature of our problem we can create a massive speedup for calculating the W costs used to direct the tree search.

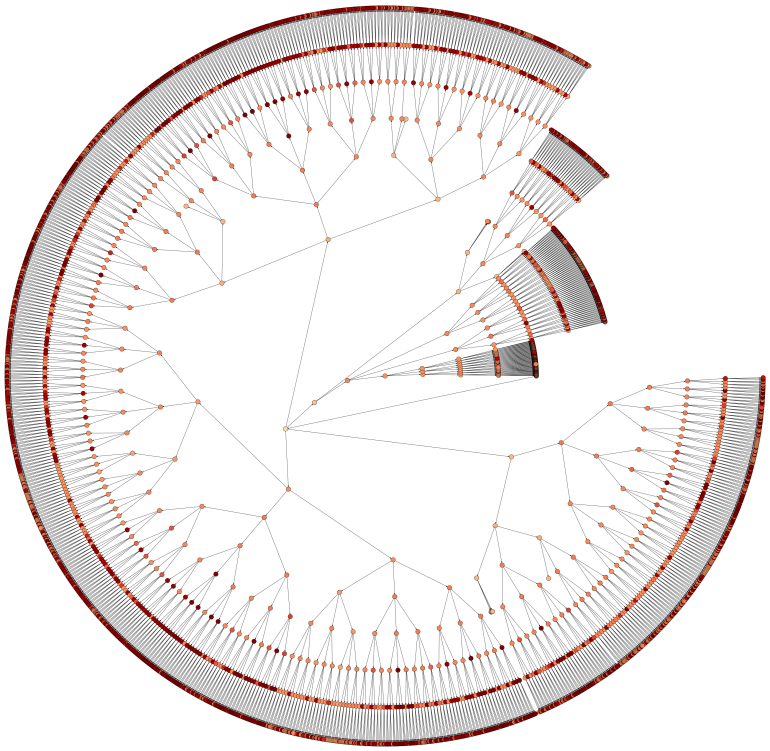

Figure 2: Largest search tree generated on AWS contained 3113 nodes, each node corresponds to 1 architecture and the color value was determined by the highest W-coefficient found over 3 learning rates ($>$ 9000 3 qubit circuits trained).

2 Background

A leading application in quantum machine learning (QML) is the design and application of quantum circuit classifier models [6]. These models train parameterized quantum circuits (PQCs) by tuning the rotational parameters passed to single qubit gates and extracting binary or multi-class labels from the expectation of Hermitian observables for a set of n qubits. However the search space for optimal circuit ansatz is large, and while hardware constraints can inform ansatz choice [5], many quantum machine learning practitioners are left with trial and error in order to find good architectures. In this whitepaper, we propose a Quantum Optimal Subarchitecture Extraction (QOSE) algorithm, that heuristically scans the space of circuit architectures for the one most suited for the problem at hand.

Optimizing architectures of neural networks is an active field of research in machine learning [3]. In [7] Optimal Subarchitecture Extraction (OSE) is proposed, and theoretical degrees for optimality are derived. This approach has been used by Amazon to optimize the architecture of the natural language processing model BERT [8, 2], significantly increasing the models speed while retaining or even increasing performance. In the quantum computing literature there are similar studies focused on "learning quantum algorithms" [1], but we emphasize the following distinction: the goal of OSE is to execute a systematic or heuristic search over an existing space of candidate structures and currently does not include any feedback about hardware connectivity restraints.

In [7] the OSE algorithm is designed to find an optimal architecture where optimality is quantified using 3 characteristics: parameter size(p), inference speed (i) and error rate (e). These are defined in terms of parameterized quantum circuit models as:

1. The parameter size $p(\cdot)$: this to be the number of trainable parameters in our classifier, it can include both arguments to rotation gates (θ) and classical weights ($\overrightarrow{w}$)

2. The inference speed $i(\cdot)$: defined as the time to execute one forward pass of the model. In [7] it is evaluated using steps, in [8] it is evaluated using the wall clock time on a CPU. *For PQCs we have the option of two parameters, the wall clock time to evaluate the cost during the optimization step (including gradient evaluation) or counting the number of two-qubit gates (CNOT or CZ).*

3. The error rate $e(\cdot)$: We provide two proxy estimates to this value: either the accuracy of label assignments over a subset $\mathcal{B} \subset \mathcal{D}$ of the data $\mathcal{D}$, or the cost value at the last step of optimization is used.

From these quantities, the quality of a given subarchitecture is defined in [7] using the *W-coefficient*:

$$\mathcal{W}(a,b) = \left(\frac{p(a) - p(b)}{p(a)}\right)\left(\frac{\hat{i}(a) - \hat{i}(b)}{\hat{i}(a)}\right)\left(\frac{1}{\hat{e}(b)}\right) \tag{1}$$

In [7], this cost is calculated with respect to the maximum point (T) i.e. a known best performing architecture, or the architecture that maximizes p, i (parameter size and inference time). Since this point is not known a priori, we will simply set $p(T)$, and $i(T)$ as free parameters of our model that will allow us to tune the sensitivity of of our algorithm.

The remaining elements needed to implement OSE are defined as follows

1. Architecture (f): A PQC classifier constructed by applying layers of rotation gates (either RX or RY) and entangling layers (using a cycle of two-qubit gates, either ZZ or CNOT gates). For the case of ZZ gates this introduces additional parameters to train.

2. Dataset (D): For this initial demonstration we use two binary classification benchmarks, the two moons dataset and the circles dataset imported from scikit-learn.

3. Weights space (W): We place no restriction on the weights space, rotation angles can take any real-value and classical weights can take any real value. However we do fix the initialization of the rotation gates to all zero values [4].

4. Search space (Ξ): The search space is defined over a fixed number of architectures which are grown as directed by the search tree.

5. Hyperparameter set (Θ): The hyperparameters associated with training are the batch size and the learning rate.

6. Maximum training steps $s > 0$: each candidate architecture is trained for a short number of optimization steps. In [8] $s = 3$, in our demonstrations we use $s = 5, 7$.

7. Selected loss ℓ. The loss function that is used to define $\hat{e}(b)$

We note that in [7] the optimization begins over the hyperparameter space Θ and then searches over architectures. In QOSE we choose an architecture first, then determine the optimal W-coefficient over the hyperparameter space.

3 Search Tree Heuristics

In [7] there is an additional parameter called ϵ which is defined as interval size ($1 \leq \epsilon \leq |\Xi|$). After the search space is constructed, it is sorted according

to parameter size, then candidate architectures are selected at ϵ-intervals. Here we note a significant departure in our implementation: rather than select architectures at regular intervals, we grow the architectures using a directed tree search through the space Ξ.

Every node of the stores the following: a string representation of the underlying circuit architecture, the maximum value of the W-coefficient, the set of weights that correspond to the maximum W-coefficient. Additionally there are optional keywords that will save the wall clock time associated with the evaluation of the W-coefficient (in local, non-distributed execution only).

The search over the architecture space and the construction of the search tree proceeds as follows:

1. A tree is initialized with a root node (`ROOT`) with associated W-coefficient 0.0, and an empty list of stored weights.

2. The set of all descendent nodes are defined by the set of unique layer architectures.

3. An edge is connected from (`ROOT`) to a descendent when the W-coefficient for the candidate circuit formed by appending the descendent to the previous architecture.

4. A minimum tree depth is specified before starting, this is the minimum depth the tree is grown until pruning is enacted

5. The pruning is implemented based on the path cost. The shortest path from `ROOT` to a leaf is computed and the lowest f fraction of branches are removed from the tree.

4 Time Complexity

According to Theorem 2 of [7], the original OSE algorithm terminates in

$$O\left(|\Xi| + |W_T^*|\left(1 + |\Theta| \cdot |B| \cdot |\Xi| \frac{1}{\epsilon \cdot s^{\frac{3}{2}}}\right)\right) \tag{2}$$

where $1 \leq \epsilon \leq |\Xi|$ and $s > 0$ are input parameters, $B \subset D$ is a section of our data we use to compute the empirical loss function ($\hat{e}(b)$ in equation 1), and $|W_T^*| = \text{argmax}_{w \in W}(|w|)$ is the cardinality of the largest weight assignment.

In our implementation of the algorithm, we set $p(T)$, $i(T)$, and $e(T)$ as free parameters. As a result, the initialization time of our algorithm is reduced from $O(|\Xi| + |W_T^*|)$ to $O(|W_T^*|)$. Additionally, we use a search tree heuristic in conjunction with a tree-pruning algorithm to search the space Ξ and select candidate architectures.

We model our search tree heuristic as a branching process. Via simulation,

we find that the effective branching factor for our search tree is 2.0382, which we set as the expectation of the model's offspring distribution.

Because each circuit is tested for quality with the W-coefficient before being selected for pruning, at each generation level of the branching process, we train and test $3Z_n$ individuals, where Z_n is the number of individuals at that depth. It is well-known from the theory of branching processes that $\mathbb{E}[Z_n] = \mu^n$, where μ is the expectation of the offspring distribution for the process. Hence, we have that the expected number of tested nodes over d generations is

$$\sum_{n=0}^{d} \mathbb{E}[3Z_n] = 3\sum_{n=0}^{d} 2.0382^n = 2.8896(2.0382^{d+1} - 1)$$

Adapting the result from [7], this tells us that training each of the candidate architectures takes

$$O\left(2.8896(2.0382^{d+1} - 1) \cdot i(f)\frac{|B| \cdot |\Theta|}{s^{\frac{3}{2}}}\right)$$

time steps. After combining and eliminating constants, this produces a time complexity of

$$O\left(|W_T^*|\left(1 + |\Theta||B|(2.0382^{d+1} - 1) \cdot \frac{1}{s^{\frac{3}{2}}}\right)\right) \tag{3}$$

steps.

Summary of keywords and tested values

Our current implementation leaves a lot of functionality open to the user. In Table 1 we summarize the keywords which have been testing during the QHACK. While several keywords are self-explanatory, we include further detail about `circuit_type` and `readout_layer` below.

The goal of QOSE is to find an optimal subarchitecture given a specific set of gates. This gate set we use to define a `circuit_type`. During this QHACK we choose two specific circuit types. The first is `schuld`: this describes a variational circuit built with ZZ (control-Z) gates, layers of X rotation gates (X), layers of Y rotation gates (Y) and layers of Z rotation gates (Z). This gate set was chosen to build architectures close to the circuit classifiers described in [6] (we include layers of Z rotation gates for completeness). The layers of control-Z gates connect the qubits in a cycle. The second is `hardware`: this is also a variational circuit ansatz build with layers of entangling and rotation gates, but now the only two qubit gates used are `CNOT` and the entangling layers connect the qubits in a simple path. The rotational gate layers are composed of either X, Y or Z gates. The main distintion between `hardware` and `schuld` is a reduction

in the number of two qubit gates, and the restrictions on qubit connectivity (to minimize SWAP gate error).

The final layer of the classifier is crucial in assigning labels from a prepared state. We tested two different final layers, called `readout_layer`. The first is `one_hot`: this builds a circuit on n qubits but returns m- $\rangle Z\langle$ expectation values (where m is the number of classes in our dataset). A model constructed using this final layer is a fully quantum classifier, there are no classical parameters.

The other layer kind is `weighted_neuron`: a circuit constructed with this final layer is built on n qubits and returns n- $\rangle Z\langle$ expectation values. These expectation values are reduced to a single label using a weighted sum: a classical weight vector $\overrightarrow{w}$ is used to define $\overrightarrow{w}^T[\rangle Z_0\langle, \rangle Z_1\langle, \ldots, \rangle Z_{n-1}\langle$ which is then passed through a nonlinear sigmoidal function.

References

[1] Lukasz Cincio et al. "Learning the quantum algorithm for state overlap". In: *New Journal of Physics* 20.11 (2018), p. 113022.

[2] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://www.aclweb.org/anthology/N19-1423.

[3] T. Elsken, J. H. Metzen, and F. Hutter. "Neural Architecture Search: A Survey". In: *ArXiv* abs/1808.05377 (2019).

[4] Edward Grant et al. "An initialization strategy for addressing barren plateaus in parametrized quantum circuits". In: *Quantum* 3 (Dec. 2019), p. 214. ISSN: 2521-327X. DOI: 10.22331/q-2019-12-09-214. URL: https://doi.org/10.22331/q-2019-12-09-214.

[5] Vojtěch Havlíček et al. "Supervised learning with quantum-enhanced feature spaces". In: *Nature* 567.7747 (2019), pp. 209–212.

[6] Maria Schuld et al. "Circuit-centric quantum classifiers". In: *Physical Review A* 101.3 (2020), p. 032308.

[7] Adrian de Wynter. *An Approximation Algorithm for Optimal Subarchitecture Extraction*. 2020. eprint: arXiv:2010.08512.

[8] Adrian de Wynter and Daniel J. Perry. *Optimal Subarchitecture Extraction For BERT*. 2020. eprint: arXiv:2010.10499.

keyword	example value	description
nqubits	4	number of qubits
`min_tree_depth`	4	minimum depth to grow before pruning
`max_tree_depth`	5	maximum depth of tree
`prune_rate`	0.15	how aggressively to prune (0.0 no pruning, 1.0 max)
`prune_step`	3	how frequently to prune
`plot_trees`	False	show plots of search tree during execution
`data_set`	("moons" or "circles")	dataset (D)
nsteps	5	number of steps to train each architecture (s)
optim	qml.AdamOptimizer	PennyLane Optimizer (note: `qml.QuantumNaturalGradient` currently not supported)
`batch_sizes`	[8, 16, 32, 64]	batch sizes
`n_samples`	1500	number of samples in dataset
`learning_rates`	[0.001, 0.005, 0.01]	learning rates
`save_frequency`	1	how frequently to save tree
`save_path`	data_path	path to directory
`save_timing`	True	save timing info for W- evaluation
`circuit_type`	('schuld' or 'hardware ')	circuit class
Tmax	[100, 100, 100]	[max params, max time, max cnots]
`inf_time`	(timeit or `num_cnots`)	surrogate for inference time
fill	(redundant or pad)	Method to build out feature vector if nqubits > features
`rate_type`	(`accuracy` or `batch_cost`)	surrogate for error rate
`readout_layer`	(`one_hot` or `weighted_neuron`)	readout layer

Table 1: Summary table of keywords tested during QHACK 2021.