QCompress Documentation

Release 0.0.1.dev11

hannahsim

Feb 12, 2019

Contents

QCompress is a Python framework for the quantum autoencoder (QAE) algorithm. Using the code, the user can execute instances of the algorithm on either a quantum simulator or a quantum processor provided by Rigetti Computing's Quantum Cloud Services. For a more in-depth description of QCompress (including the naming convention for the types of qubits involved in the QAE circuit), please go to section *Introduction to QAE and QCompress*.

For more information about the algorithm, see Romero et al. Note that we deviate from the training technique used in the original paper and instead introduce two alternative autoencoder training schemes that require lower-depth circuits (see Sim et al).

CHAPTER 1

Features

This code is based on an older version written during Rigetti Computing's hackathon in April 2018. Since then, we've updated and enhanced the code, supporting the following features:

- Executability on Rigetti's quantum processor(s)
- Several training schemes for the autoencoder
- Use of the `RESET` operation for the encoding qubits (lowers qubit requirement)
- User-definable training circuit and/or classical optimization routine

CHAPTER 2

Contents

2.1 Installing QCompress

There are a few options for installing QCompress:

1. To install QCompress using `pip`, execute:

```
pip install qcompress
```

2. To install QCompress using `conda`, execute:

```
conda install -c rigetti -c hsim13372 qcompress
```

3. To instead install QCompress from source, clone this repository, `cd` into it, and run:

```
git clone https://github.com/hsim13372/QCompress
cd QCompress
python -m pip install -e .
```

Try executing `import qcompress` to test the installation in your terminal.

Note that the pyQuil version used requires Python 3.6 or later.

2.1.1 Installing QCompress on QMI

For installing QCompress on a user's Quantum Machine Image (QMI), we recommend the following steps:

1. Connect to your QMI with SSH.
2. Launch a Python virtual environment:

```
source ~/pyquil/venv/bin/activate
```

3. To install QCompress, clone then install from github or install using `pip`:

```
git clone https://github.com/hsim13372/QCompress
cd QCompress
pip install -e .
```

or

```
pip install qcompress
```

4. To execute the Jupyter notebook demos or run QCompress on Jupyter notebooks in general, execute:

```
tmux new -s <ENTER-SESSION-NAME>
source ~/pyquil/venv/bin/activate

pip install jupyter
cd <ENTER-DIRECTORY-FOR-NOTEBOOK>
jupyter notebook
```

5. (Optional) To run your quantum autoencoder instance on the QPU, book reservations in the compute schedule via `qcs reserve`.

NOTE: We assume the user has already set up his/her QMI. If the user is new to QCS, please refer to Rigetti QCS docs to get started!

2.2 Introduction to QAE and QCompress

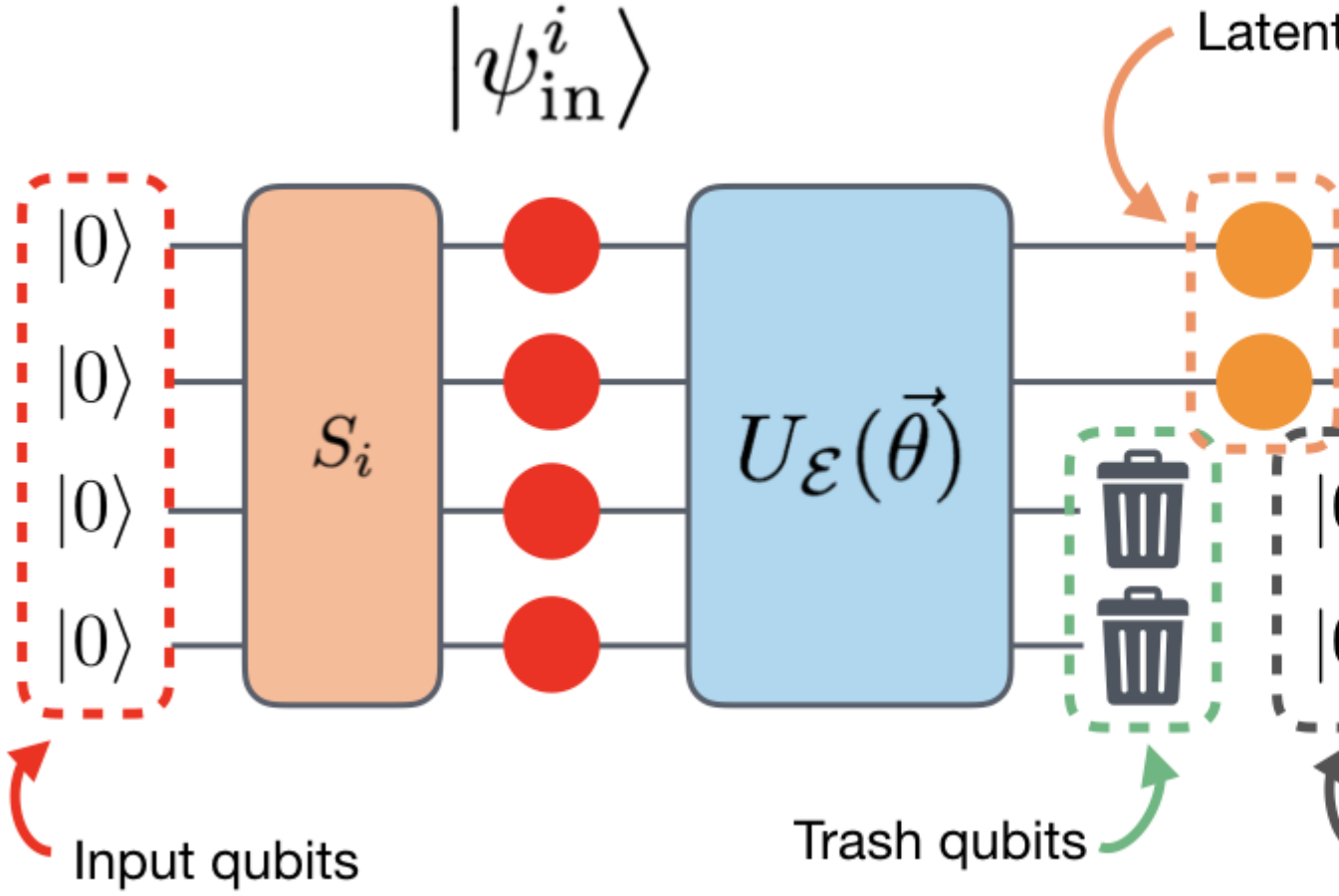
2.2.1 What is a quantum autoencoder (QAE)?

Similar to the idea of classical autoencoders, a quantum autoencoder is a function whose parameters are optimized across a training data such that given an -qubit input , the autoencoder attempts to reproduce . Part of this process involves expressing the input data set using a fewer number of qubits (using qubits out of). This means that if the QAE is successfully trained, the corresponding circuit represents a compressed encoding of the input , which may be useful to applications such as dimension reduction of quantum data. For a more in-depth explanation of the QAE, please refer to the original paper by Romero et al. In addition, we note that this is one possible realization of a "quantum" autoencoder and that there are other proposed models for the quantum autoencoder.

2.2.2 QAE model in QCompress

We note that our setup of the quantum autoencoder in QCompress is different from what was proposed by Romero et al. In the original paper, the protocol includes a SWAP test to measure the overlap between the "reference" and "trash" states. However, implementing the SWAP test is generally expensive for today's quantum processors. Instead, we implement two alternative training schemes, described in Sim et al.

Before going into the details, we use the following naming conventions for the types of qubits involved in the QAE model in QCompress:



In the current version of QCompress, there are two main training schemes:

1. **Halfway training** (or trash training) - In this scheme, we execute only the state preparation followed by the training circuit and count the probability of measure all 0's on the "trash" qubits (i.e. input qubits that are **not** the latent space qubits).
2. **Full training** - In this scheme, we execute the entire circuit (state preparation, training, un-training, un-state preparation) and count the probability of measuring all 0's on the "output" qubits. There are two possible

sub-strategies:

2a. **Full training with reset**: With the `RESET` feature in pyQuil, we reset the input qubits (except the latent space qubit) such that these qubits **are** the refresh qubits in the latter half of the QAE circuit. Therefore, in total, this method requires qubits.

2b. **Full training without reset**: Without the reset feature, we introduce new qubits for the refresh qubits. Therefore, in total, this method requires qubits.

NOTE: For the loss function, we average over the training set losses and negate the value to cast as a minimization problem.

2.2.3 Overview of QCompress

Here, we provide a high-level overview of how to prepare and execute an instance of the QAE algorithm using QCompress. The major steps involved are:

1. Prepare quantum data: generate state preparation circuits, for each data point .
2. Select a parametrized circuit to train the QAE.
3. Initialize the QAE instance.
4. Set up the Forest connection: this is where the user can decide on executing the instance on the simulator or the actual quantum device on Rigetti's QCS.
5. Split data set into training and test sets.
6. Set initial guess for the parameters, and train the QAE.
7. Evaluate the QAE performance by predicting against the test set.

2.2.4 Examples

We provide several Jupyter notebooks to demonstrate the utility of QCompress. We recommend going through the notebooks in the order shown in the table (top-down).

Notebook	Feature(s)
qae_h2_demo.ipynb	Simulates the compression of the ground states of the hydrogen molecule. Uses OpenFermion and grove to generate data. Demonstrates the "halfway" training scheme.
qae_two_qubit_demo.ipynb	Simulates the compression of a two-qubit data set. Outlines how to run an instance on an actual device. Demonstrates the "full with reset" training scheme.
run_landscape_scan.ipynb	Shows user how to run landscape scans for small (few-parameter) instances. Demonstrates setup of the "full with no reset" training scheme.

2.2.5 How to cite QCompress

When using QCompress for research projects, please cite:

> Sukin Sim, Yudong Cao, Jonathan Romero, Peter D. Johnson and Alán Aspuru-Guzik. *A framework for algorithm deployment on cloud-based quantum computers*. arXiv:1810.10576. 2018.

2.3 Demo: Compressing ground states of molecular hydrogen

In this demo, we will try to compress ground states of molecular hydrogen at various bond lengths. We start with expressing each state using 4 qubits and try to compress the information to 1 qubit (i.e. implement a 4-1-4 quantum autoencoder).

In the following section, we review both the full and halfway training schemes. However, in the notebook, we execute the halfway training case.

2.3.1 Background: Full cost function training

The QAE circuit for full cost function training looks like the following:

We note that our setup is different from what was proposed in the original paper. As shown in the figure above, we use 7 total qubits for the 4-1-4 autoencoder, using the last 3 qubits (qubits , , and) as refresh qubits. The unitary represents the state preparation circuit, gates implemented to produce the input data set. The unitary represents the training circuit that will be responsible for representing the data set using a fewer number of qubits, in this case using a single qubit. The tilde symbol above the daggered operations indicates that the qubit indexing has been adjusted such that , , , and . For clarity, refer to the figure below for an equivalent circuit with the refresh qubits moved around. So qubit is to be the "latent space qubit," or qubit to hold the compressed information. Using the circuit structure above (applying and then effectively **un**-applying and), we train the autoencoder by propagating the QAE circuit with proposed parameters and computing the probability of obtaining measurements of 0000 for the latent space and refresh qubits (to). We negate this value for casting as a minimization problem and average over the training set to compute a single loss value.

2.3.2 Background: Halfway cost function training

In the halfway cost function training case, the circuit looks like the following:

Here, the cost function is the negated probability of obtaining the measurement 000 for the trash qubits.

2.3.3 Demo Outline

We break down this demo into the following steps: 1. Preparation of the quantum data 2. Initializing the QAE 2. Setting up the Forest connection 3. Dividing the dataset into training and test sets 4. Training the network 5. Testing the network

NOTE: While the QCompress framework was developed to execute on both the QVM and the QPU (i.e. simulator and quantum device, respectively), this particular demo runs a simulation (i.e. uses QVM).

Let us begin! **Note that this tutorial requires installation of 'OpenFermion <https://github.com/quantumlib/OpenFermion>'__!**

[1]:
```
# Import modules
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os

from openfermion.hamiltonians import MolecularData
from openfermion.transforms import get_sparse_operator, jordan_wigner
from openfermion.utils import get_ground_state

from pyquil.api import WavefunctionSimulator
from pyquil.gates import *
```

(continues on next page)

(continued from previous page)

```
from pyquil import Program

import demo_utils

from qcompress.qae_engine import *
from qcompress.utils import *
from qcompress.config import DATA_DIRECTORY

global pi
pi = np.pi
```

2.3.4 QAE Settings

In the cell below, we enter the settings for the QAE.

NOTE: Because QCompress was designed to run on the quantum device (as well as the simulator), we need to anticipate nontrival mappings between abstract qubits and physical qubits. The dictionaries `q_in`, `q_latent`, and `q_refresh` are abstract-to-physical qubit mappings for the input, latent space, and refresh qubits respectively. A cool plug-in/feature to add would be to have an automated "qubit mapper" to determine the optimal or near-optimal abstract-to-physical qubit mappings for a particular QAE instance.

In this simulation, we will skip the circuit compilation step by turning off `compile_program`.

[2]:
```
### QAE setup options

# Abstract-to-physical qubit mapping
q_in = {'q0': 0, 'q1': 1, 'q2': 2, 'q3': 3} # Input qubits
q_latent = {'q3': 3}                        # Latent space qubits
q_refresh = None

# Training scheme: Halfway
trash_training = True

# Simulator settings
cxn_setting = '4q-qvm'
compile_program = False
n_shots = 3000
```

2.3.5 Qubit labeling

In the cell below, we produce lists of **ordered** physical qubit indices involved in the compression and recovery maps of the quantum autoencoder. Depending on the training and reset schemes, we may use different qubits for the compression vs. recovery.

Since we're employing the halfway training scheme, we don't need to assign the qubit labels for the recovery process.

[3]:
```
compression_indices = order_qubit_labels(q_in).tolist()

if not trash_training:
    q_out = merge_two_dicts(q_latent, q_refresh)
    recovery_indices = order_qubit_labels(q_out).tolist()

    if not reset:
        recovery_indices = recovery_indices[::-1]
```

(continues on next page)

(continued from previous page)

```
print("Physical qubit indices for compression : {0}".format(compression_indices))
```

```
Physical qubit indices for compression : [0, 1, 2, 3]
```

2.3.6 Generating input data

We use routines from `OpenFermion`, `forestopenfermion`, and `grove` to generate the input data set. We've provided the molecular data files for you, which were generated using `OpenFermion`'s plugin `OpenFermion-Psi4`.

[4]:
```
qvm = WavefunctionSimulator()

# MolecularData settings
molecule_name = "H2"
basis = "sto-3g"
multiplicity = "singlet"
dist_list = np.arange(0.2, 4.2, 0.1)

# Lists to store HF and FCI energies
hf_energies = []
fci_energies = []
check_energies = []

# Lists to store state preparation circuits
list_SP_circuits = []
list_SP_circuits_dag = []

for dist in dist_list:

    # Fetch file path
    dist = "{0:.1f}".format(dist)
    file_path = os.path.join(DATA_DIRECTORY, "{0}_{1}_{2}_{3}.hdf5".format(molecule_
↪name,
                                                                           basis,
                                                                           multiplicity,
                                                                           dist))

    # Extract molecular info
    molecule = MolecularData(filename=file_path)
    n_qubits = molecule.n_qubits
    hf_energies.append(molecule.hf_energy)
    fci_energies.append(molecule.fci_energy)
    molecular_ham = molecule.get_molecular_hamiltonian()

    # Set up hamiltonian in qubit basis
    qubit_ham = jordan_wigner(molecular_ham)

    # Convert from OpenFermion's to PyQuil's data type (QubitOperator to PauliTerm/
↪PauliSum)
    qubit_ham_pyquil = demo_utils.qubitop_to_pyquilpauli(qubit_ham)

    # Sanity check: Obtain ground state energy and check with MolecularData's FCI␣
↪energy
    molecular_ham_sparse = get_sparse_operator(operator=molecular_ham, n_qubits=n_
↪qubits)
```

(continues on next page)

(continued from previous page)

```
    ground_energy, ground_state = get_ground_state(molecular_ham_sparse)
    assert np.isclose(molecule.fci_energy, ground_energy)

    # Generate unitary to prepare ground states
    state_prep_unitary = demo_utils.create_arbitrary_state(
                            ground_state,
                            qubits=compression_indices)

    if not trash_training:
        if reset:
            # Generate daggered state prep unitary (WITH NEW/ADJUSTED INDICES!)
            state_prep_unitary_dag = demo_utils.create_arbitrary_state(
                                        ground_state,
                                        qubits=compression_indices).dagger()
        else:
            # Generate daggered state prep unitary (WITH NEW/ADJUSTED INDICES!)
            state_prep_unitary_dag = demo_utils.create_arbitrary_state(
                                        ground_state,
                                        qubits=recovery_indices).dagger()

    # Sanity check: Compute energy wrt wavefunction evolved under state_prep_unitary
    wfn = qvm.wavefunction(state_prep_unitary)
    ket = wfn.amplitudes
    bra = np.transpose(np.conjugate(wfn.amplitudes))
    ham_matrix = molecular_ham_sparse.toarray()
    energy_expectation = np.dot(bra, np.dot(ham_matrix, ket))
    check_energies.append(energy_expectation)

    # Store circuits
    list_SP_circuits.append(state_prep_unitary)
    if not trash_training:
        list_SP_circuits_dag.append(state_prep_unitary_dag)
```

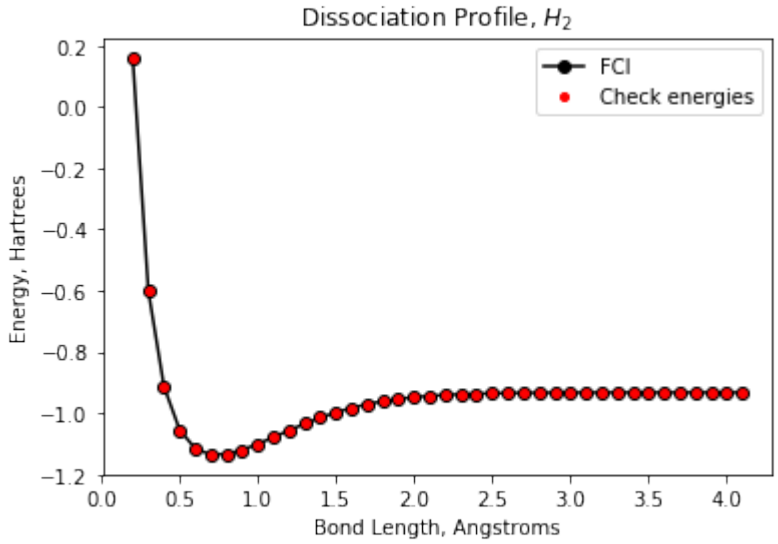
[ ]:

Plotting the energies of the input data set

To (visually) check our state preparation circuits, we run these circuits and plot the energies. The "check" energies overlay nicely with the FCI energies.

[5]:
```
imag_components = np.array([E.imag for E in check_energies])
assert np.isclose(imag_components, np.zeros(len(imag_components))).all()
check_energies = [E.real for E in check_energies]

plt.plot(dist_list, fci_energies, 'ko-', markersize=6, label='FCI')
plt.plot(dist_list, check_energies, 'ro', markersize=4, label='Check energies')
plt.title("Dissociation Profile, $H_2$")
plt.xlabel("Bond Length, Angstroms")
plt.ylabel("Energy, Hartrees")
plt.legend()
plt.show()
```



[ ]:

2.3.7 Training circuit for QAE

Now we want to choose a parametrized circuit with which we hope to train to compress the input quantum data set.

For this demonstration, we use a simple two-parameter circuit, as shown below.

NOTE: For more general data sets (and general circuits), we may need to run multiple instances of the QAE with different initial guesses to find a good compression circuit.

[6]:
```
def _training_circuit(theta, qubit_indices):
    """
    Returns parametrized/training circuit.

    :param theta: (list or numpy.array, required) Vector of training parameters
    :param qubit_indices: (list, required) List of qubit indices
    :returns: Training circuit
    :rtype: pyquil.quil.Program
    """
    circuit = Program()
    circuit.inst(Program(RX(theta[0], qubit_indices[2]),
                         RX(theta[1], qubit_indices[3])))
    circuit.inst(Program(CNOT(qubit_indices[2], qubit_indices[0]),
                         CNOT(qubit_indices[3], qubit_indices[1]),
                         CNOT(qubit_indices[3], qubit_indices[2])))
    return circuit

def _training_circuit_dag(theta, qubit_indices):
    """
    Returns the daggered parametrized/training circuit.

    :param theta: (list or numpy.array, required) Vector of training parameters
    :param qubit_indices: (list, required) List of qubit indices
    :returns: Daggered training circuit
    :rtype: pyquil.quil.Program
```

(continues on next page)

(continued from previous page)

```
    """
    circuit = Program()
    circuit.inst(Program(CNOT(qubit_indices[3], qubit_indices[2]),
                         CNOT(qubit_indices[3], qubit_indices[1]),
                         CNOT(qubit_indices[2], qubit_indices[0])))
    circuit.inst(Program(RX(-theta[1], qubit_indices[3]),
                         RX(-theta[0], qubit_indices[2])))
    return circuit
```

[7]:
```
training_circuit = lambda param : _training_circuit(param, compression_indices)

if not trash_training:
    if reset:
        training_circuit_dag = lambda param : _training_circuit_dag(param,
                                                                    compression_
↪indices)
    else:
        training_circuit_dag = lambda param : _training_circuit_dag(param,
                                                                    recovery_indices)
```

[ ]:
```
```

2.3.8 Initialize the QAE engine

Here we create an instance of the `quantum_autoencoder` class.

Leveraging the features of the `Forest` platform, this quantum autoencoder "engine" allows you to run a noisy version of the QVM to get a sense of how the autoencoder performs under noise (but qvm is noiseless in this demo). In addition, the user can also run this instance on the quantum device (assuming the user is given access to one of Rigetti Computing's available QPUs).

[8]:
```
qae = quantum_autoencoder(state_prep_circuits=list_SP_circuits,
                          training_circuit=training_circuit,
                          q_in=q_in,
                          q_latent=q_latent,
                          q_refresh=q_refresh,
                          trash_training=trash_training,
                          compile_program=compile_program,
                          n_shots=n_shots,
                          print_interval=1)
```

After defining the instance, we set up the Forest connection (in this case, a simulator).

[9]:
```
qae.setup_forest_cxn(cxn_setting)
```

Let's split the data set into training and test set. If we don't input the argument `train_indices`, the data set will be randomly split. However, knowing our quantum data set, we may want to choose various regions along the PES (the energy curve shown above) to train the entire function. Here, we pick 6 out of 40 data points for our training set.

[10]:
```
qae.train_test_split(train_indices=[3, 10, 15, 20, 30, 35])
```

Let's print some information about the QAE instance.

[11]:
```
print(qae)
```

```
QCompress Setting
=================
QAE type: 4-1-4
Data size: 40
Training set size: 6
Training mode: halfway cost function
Compile program: False
Forest connection: 4q-qvm
  Connection type: QVM
```

[ ]:
```
```

2.3.9 Training

The autoencoder is trained in the cell below, where the default optimization algorithm is Constrained Optimization BY Linear Approximation (COBYLA). The lowest possible mean loss value is -1.000.

[12]:
```
%%time

initial_guess = [pi/2., 0.]
avg_loss_train = qae.train(initial_guess)
```

```
Iter    0 Mean Loss: -0.0000000
Iter    1 Mean Loss: -0.0000000
Iter    2 Mean Loss: -0.0011667
Iter    3 Mean Loss: -0.0043333
Iter    4 Mean Loss: -0.0111667
Iter    5 Mean Loss: -0.0157778
Iter    6 Mean Loss: -0.0258889
Iter    7 Mean Loss: -0.0393889
Iter    8 Mean Loss: -0.0510556
Iter    9 Mean Loss: -0.0647778
Iter   10 Mean Loss: -0.0860556
Iter   11 Mean Loss: -0.1017778
Iter   12 Mean Loss: -0.1266111
Iter   13 Mean Loss: -0.1475000
Iter   14 Mean Loss: -0.1716667
Iter   15 Mean Loss: -0.2078889
Iter   16 Mean Loss: -0.2444444
Iter   17 Mean Loss: -0.2815000
Iter   18 Mean Loss: -0.3136667
Iter   19 Mean Loss: -0.3497778
Iter   20 Mean Loss: -0.3960556
Iter   21 Mean Loss: -0.4407778
Iter   22 Mean Loss: -0.4816667
Iter   23 Mean Loss: -0.5263333
Iter   24 Mean Loss: -0.5710556
Iter   25 Mean Loss: -0.6225000
Iter   26 Mean Loss: -0.6395556
Iter   27 Mean Loss: -0.6777778
Iter   28 Mean Loss: -0.7205556
Iter   29 Mean Loss: -0.7716667
Iter   30 Mean Loss: -0.7977222
Iter   31 Mean Loss: -0.8250556
```

(continues on next page)

(continued from previous page)

```
Iter   32 Mean Loss: -0.8601111
Iter   33 Mean Loss: -0.8893333
Iter   34 Mean Loss: -0.9110556
Iter   35 Mean Loss: -0.9350556
Iter   36 Mean Loss: -0.9600000
Iter   37 Mean Loss: -0.9718333
Iter   38 Mean Loss: -0.9843333
Iter   39 Mean Loss: -0.9901111
Iter   40 Mean Loss: -0.9892222
Iter   41 Mean Loss: -0.9955556
Iter   42 Mean Loss: -0.9981111
Iter   43 Mean Loss: -0.9992222
Iter   44 Mean Loss: -1.0000000
Iter   45 Mean Loss: -0.9997222
Iter   46 Mean Loss: -0.9996667
Iter   47 Mean Loss: -1.0000000
Iter   48 Mean Loss: -1.0000000
Iter   49 Mean Loss: -1.0000000
Iter   50 Mean Loss: -1.0000000
Iter   51 Mean Loss: -1.0000000
Iter   52 Mean Loss: -1.0000000
Mean loss for training data: -1.0
CPU times: user 3.74 s, sys: 78.5 ms, total: 3.82 s
Wall time: 2min 17s
```

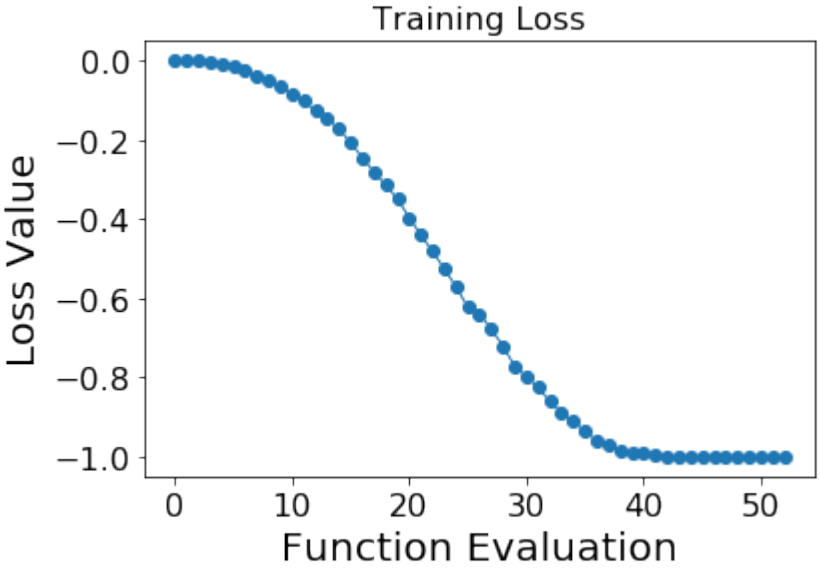
Printing the optimized parameters

[13]:
```
print(qae.optimized_params)
```

```
[3.1337883  3.14026142]
```

Plot training losses

[14]:
```
fig = plt.figure(figsize=(6, 4))
plt.plot(qae.train_history, 'o-', linewidth=1)
plt.title("Training Loss", fontsize=16)
plt.xlabel("Function Evaluation",fontsize=20)
plt.ylabel("Loss Value", fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.show()
```



2.3.10 Testing

Now test the optimized network against the rest of the data set (i.e. use the optimized parameters to try to compress then recover each test data point).

[15]:
```
avg_loss_test = qae.predict()
```

```
Iter   53 Mean Loss: -0.9999804
Mean loss for test data: -0.9999803921568627
```

[ ]:
```
```

2.4 Demo: Two-qubit QAE instance

In this demo, we compress a two-qubit data set such that we have its (lossy) description using one qubit.

We show the state preparation and training circuits below (circuits for both training schemes are shown but in this notebook, we run the "full with reset" method):

We first generate the data set by varying (40 equally-spaced points from to). Then, a single-parameter circuit is used to find the 2-1-2 map. Looking at the circuit, the minimum should be when .

[1]:
```
# Import modules
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
import scipy.optimize

from pyquil.gates import *
from pyquil import Program
```

(continues on next page)

(continued from previous page)

```
from qcompress.qae_engine import *
from qcompress.utils import *

global pi
pi = np.pi
```

2.4.1 QAE Settings

In the cell below, we enter the settings for the QAE.

NOTE: Because QCompress was designed to run on the quantum device (as well as the simulator), we need to anticipate nontrival mappings between abstract qubits and physical qubits. The dictionaries `q_in`, `q_latent`, and `q_refresh` are abstract-to-physical qubit mappings for the input, latent space, and refresh qubits respectively. A cool plug-in/feature to add would be to have an automated "qubit mapper" to determine the optimal or near-optimal abstract-to-physical qubit mappings for a particular QAE instance.

[2]:
```
### QAE setup options

# Abstract-to-physical qubit mapping
q_in = {'q0': 0, 'q1': 1} # Input qubits
q_latent = {'q1': 1}      # Latent space qubits
q_refresh = {'q0': 0}     # Refresh qubits

# Training scheme: Full with reset feature (q_refresh = q_in - q_latent)
trash_training = False
reset = True

# Simulator settings
cxn_setting = '2q-qvm'
n_shots = 5000
```

Aside: Running on the QPU

To execute the quantum autoencoder on an actual quantum device, the user simply replaces `cxn_setting` to a valid quantum processing unit (QPU) setting. This is also assuming the user has already made reservations on his/her quantum machine image (QMI) to use the QPU. To sign up for an account on Rigetti's Quantum Cloud Services (QCS), click here.

2.4.2 Data preparation circuits

To prepare the quantum data, we define the state preparation circuits (and their daggered circuits). In this particular example, we will generate the data by scanning over various values of `phi`.

[3]:
```
def _state_prep_circuit(phi, qubit_indices):
    """
    Returns parametrized state preparation circuit.
    We will vary over phi to generate the data set.

    :param phi: (list or numpy.array, required) List or array of data generation␣
↪parameters
    :param qubit_indices: (list, required) List of qubit indices
    :returns: State preparation circuit
```

(continues on next page)

(continued from previous page)

```
    :rtype: pyquil.quil.Program
    """
    circuit = Program()
    circuit += Program(RY(phi[0], qubit_indices[1]))
    circuit += Program(CNOT(qubit_indices[1], qubit_indices[0]))
    return circuit

def _state_prep_circuit_dag(phi, qubit_indices):
    """
    Returns the daggered version of the state preparation circuit.

    :param phi: (list or numpy.array, required) List or array of data generation␣
↪parameters
    :param qubit_indices: (list, required) List of qubit indices
    :returns: State un-preparation circuit
    :rtype: pyquil.quil.Program
    """
    circuit = Program()
    circuit += Program(CNOT(qubit_indices[1], qubit_indices[0]))
    circuit += Program(RY(-phi[0], qubit_indices[1]))
    return circuit
```

2.4.3 Qubit labeling

In the cell below, we produce lists of **ordered** physical qubit indices involved in the compression and recovery maps of the quantum autoencoder. Depending on the training and reset schemes, we may use different qubits for the compression vs. recovery.

[4]:
```
compression_indices = order_qubit_labels(q_in).tolist()

q_out = merge_two_dicts(q_latent, q_refresh)
recovery_indices = order_qubit_labels(q_out).tolist()

if not reset:
    recovery_indices = recovery_indices[::-1]

print("Physical qubit indices for compression : {0}".format(compression_indices))
print("Physical qubit indices for recovery    : {0}".format(recovery_indices))
```

```
Physical qubit indices for compression : [0, 1]
Physical qubit indices for recovery    : [0, 1]
```

For the full training scheme with no resetting feature, this will require the three total qubits.

The first two qubits (`q0`, `q1`) will be used to encode the quantum data. `q1` will then be used as the latent space qubit, meaning our objective will be to reward the training conditions that "push" the information to the latent space qubit. Then, a refresh qubit, `q2`, is added to recover the original data.

2.4.4 Data generation

After determining the qubit mapping, we add this physical qubit information to the state preparation circuits and store the "mapped" circuits.

[5]:
```
# Lists to store state preparation circuits
list_SP_circuits = []
list_SP_circuits_dag = []

phi_list = np.linspace(-pi/2., pi/2., 40)

for angle in phi_list:

    # Map state prep circuits
    state_prep_circuit = _state_prep_circuit([angle], compression_indices)

    # Map daggered state prep circuits
    if reset:
        state_prep_circuit_dag = _state_prep_circuit_dag([angle], compression_indices)
    else:
        state_prep_circuit_dag = _state_prep_circuit_dag([angle], recovery_indices)

    # Store mapped circuits
    list_SP_circuits.append(state_prep_circuit)
    list_SP_circuits_dag.append(state_prep_circuit_dag)
```

2.4.5 Training circuit preparation

In this step, we choose a parametrized quantum circuit that will be trained to compress then recover the input data set.

NOTE: This is a simple one-parameter training circuit.

[6]:
```
def _training_circuit(theta, qubit_indices):
    """
    Returns parametrized/training circuit.

    :param theta: (list or numpy.array, required) Vector of training parameters
    :param qubit_indices: (list, required) List of qubit indices
    :returns: Training circuit
    :rtype: pyquil.quil.Program
    """
    circuit = Program()
    circuit += Program(RY(-theta[0]/2, qubit_indices[0]))
    circuit += Program(CNOT(qubit_indices[1], qubit_indices[0]))
    return circuit

def _training_circuit_dag(theta, qubit_indices):
    """
    Returns the daggered parametrized/training circuit.

    :param theta: (list or numpy.array, required) Vector of training parameters
    :param qubit_indices: (list, required) List of qubit indices
    :returns: Daggered training circuit
    :rtype: pyquil.quil.Program
    """
    circuit = Program()
    circuit += Program(CNOT(qubit_indices[1], qubit_indices[0]))
    circuit += Program(RY(theta[0]/2, qubit_indices[0]))
    return circuit
```

As was done for the state preparation circuits, we also map the training circuits with physical qubits we want to use.

[7]:
```
training_circuit = lambda param : _training_circuit(param, compression_indices)

if reset:
    training_circuit_dag = lambda param : _training_circuit_dag(param, compression_
↪indices)
else:
    training_circuit_dag = lambda param : _training_circuit_dag(param, recovery_
↪indices)
```

2.4.6 Define the QAE instance

Here, we initialize a QAE instance. This is where the user can decide which optimizer to use, etc.

For this demo, we use `scipy`'s POWELL optimizer. Because various optimizers have different output variable names, we allow the user to enter a function that parses the optimization output. This function always returns the optimized parameter then its function value (in this order). We show an example of how to use this feature below (see `opt_result_parse` function). The POWELL optimizer returns a list of output values, in which the first and second elements are the optimized parameters and their corresponding function value, respectively.

[8]:
```
minimizer = scipy.optimize.fmin_powell
minimizer_args = []
minimizer_kwargs = ({'xtol': 0.0001, 'ftol': 0.0001, 'maxiter': 500,
                     'full_output': 1, 'retall': 1})
opt_result_parse = lambda opt_res: ([opt_res[0]], opt_res[1])
```

[9]:
```
qae = quantum_autoencoder(state_prep_circuits=list_SP_circuits,
                          training_circuit=training_circuit,
                          q_in=q_in,
                          q_latent=q_latent,
                          q_refresh=q_refresh,
                          state_prep_circuits_dag=list_SP_circuits_dag,
                          training_circuit_dag=training_circuit_dag,
                          trash_training=trash_training,
                          reset=reset,
                          minimizer=minimizer,
                          minimizer_args=minimizer_args,
                          minimizer_kwargs=minimizer_kwargs,
                          opt_result_parse=opt_result_parse,
                          n_shots=n_shots,
                          print_interval=1)
```

After defining the instance, we set up the Forest connection (in this case, a simulator) and split the data set.

[10]:
```
qae.setup_forest_cxn(cxn_setting)
```

[11]:
```
qae.train_test_split(train_indices=[1, 31, 16, 7, 20, 23, 9, 17])
```

[12]:
```
print(qae)
```

```
QCompress Setting
=================
QAE type: 2-1-2
Data size: 40
Training set size: 8
Training mode: full cost function
```

(continues on next page)

(continued from previous page)

```
  Reset qubits: True
Compile program: False
Forest connection: 2q-qvm
  Connection type: QVM
```

2.4.7 Training

The autoencoder is trained in the cell below. The lowest possible mean loss value is -1.000.

[13]:
```
%%time
initial_guess = [pi/1.2]

avg_loss_train = qae.train(initial_guess)
```

```
Iter    0 Mean Loss: -0.5382250
Iter    1 Mean Loss: -0.5400500
Iter    2 Mean Loss: -0.2839000
Iter    3 Mean Loss: -0.9166500
Iter    4 Mean Loss: -0.8539250
Iter    5 Mean Loss: -0.9167750
Iter    6 Mean Loss: -0.2814000
Iter    7 Mean Loss: -0.9488000
Iter    8 Mean Loss: -0.9997750
Iter    9 Mean Loss: -1.0000000
Iter   10 Mean Loss: -0.9999000
Iter   11 Mean Loss: -0.9999500
Iter   12 Mean Loss: -0.5419250
Iter   13 Mean Loss: -1.0000000
Iter   14 Mean Loss: -0.5503500
Iter   15 Mean Loss: -0.1691000
Iter   16 Mean Loss: -1.0000000
Iter   17 Mean Loss: -0.7972250
Iter   18 Mean Loss: -0.9126500
Iter   19 Mean Loss: -0.9996750
Iter   20 Mean Loss: -0.9999500
Iter   21 Mean Loss: -1.0000000
Iter   22 Mean Loss: -1.0000000
Iter   23 Mean Loss: -1.0000000
Iter   24 Mean Loss: -1.0000000
Iter   25 Mean Loss: -1.0000000
Iter   26 Mean Loss: -1.0000000
Iter   27 Mean Loss: -1.0000000
Iter   28 Mean Loss: -1.0000000
Iter   29 Mean Loss: -1.0000000
Iter   30 Mean Loss: -1.0000000
Iter   31 Mean Loss: -1.0000000
Iter   32 Mean Loss: -1.0000000
Iter   33 Mean Loss: -1.0000000
Iter   34 Mean Loss: -1.0000000
Iter   35 Mean Loss: -1.0000000
Iter   36 Mean Loss: -1.0000000
Iter   37 Mean Loss: -1.0000000
Iter   38 Mean Loss: -1.0000000
Iter   39 Mean Loss: -1.0000000
Iter   40 Mean Loss: -1.0000000
```

(continues on next page)

(continued from previous page)

```
Iter   41 Mean Loss: -1.0000000
Iter   42 Mean Loss: -1.0000000
Iter   43 Mean Loss: -1.0000000
Iter   44 Mean Loss: -1.0000000
Iter   45 Mean Loss: -1.0000000
Iter   46 Mean Loss: -1.0000000
Iter   47 Mean Loss: -1.0000000
Iter   48 Mean Loss: -1.0000000
Iter   49 Mean Loss: -1.0000000
Iter   50 Mean Loss: -1.0000000
Iter   51 Mean Loss: -1.0000000
Iter   52 Mean Loss: -1.0000000
Iter   53 Mean Loss: -1.0000000
Iter   54 Mean Loss: -1.0000000
Iter   55 Mean Loss: -1.0000000
Iter   56 Mean Loss: -1.0000000
Iter   57 Mean Loss: -1.0000000
Iter   58 Mean Loss: -1.0000000
Iter   59 Mean Loss: -1.0000000
Iter   60 Mean Loss: -1.0000000
Iter   61 Mean Loss: -1.0000000
Iter   62 Mean Loss: -1.0000000
Iter   63 Mean Loss: -1.0000000
Iter   64 Mean Loss: -1.0000000
Iter   65 Mean Loss: -1.0000000
Iter   66 Mean Loss: -1.0000000
Iter   67 Mean Loss: -1.0000000
Iter   68 Mean Loss: -1.0000000
Iter   69 Mean Loss: -1.0000000
Iter   70 Mean Loss: -1.0000000
Iter   71 Mean Loss: -1.0000000
Iter   72 Mean Loss: -1.0000000
Optimization terminated successfully.
         Current function value: -1.000000
         Iterations: 2
         Function evaluations: 73
Mean loss for training data: -1.0
CPU times: user 6 s, sys: 135 ms, total: 6.14 s
Wall time: 57.3 s
```

Printing the optimized parameters

[14]:
```
print(qae.optimized_params)
```

```
[array(0.00155588)]
```

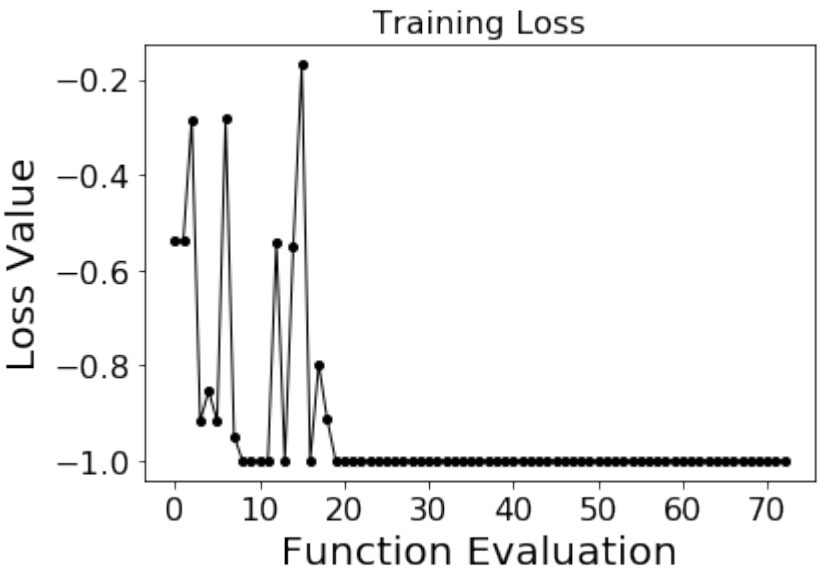
Plot training loss

[18]:
```
# Visualize loss across function evaluations
fig = plt.figure(figsize=(6, 4))
plt.plot(qae.train_history, 'ko-', markersize=4, linewidth=1)
plt.title("Training Loss", fontsize=16)
plt.xlabel("Function Evaluation",fontsize=20)
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("Loss Value", fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.show()
```



2.4.8 Testing

Now test the optimized network against the rest of the data set (i.e. use the optimized parameters to try to compress then recover each test data point).

[16]:
```
avg_loss_test = qae.predict()
```

```
Iter   73 Mean Loss: -1.0000000
Mean loss for test data: -1.0
```

[ ]:
```
```

2.5 Demo: Running parameter scans

For small examples, i.e. autoencoder instances that utilize training circuits with a small number of parameters, we can plot and visualize loss landscapes by scanning over the circuit parameters.

For this demonstration, we use the two-parameter example shown in qae_two_qubit_demo.ipynb. Let's first set up this instance.

[1]:
```
# Import modules
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
import scipy.optimize
```

(continues on next page)

(continued from previous page)

```
from pyquil.gates import *
from pyquil import Program

from qcompress.qae_engine import *
from qcompress.utils import *

global pi
pi = np.pi
```

2.5.1 QAE Settings

In the cell below, we enter the settings for the QAE. This two-qubit instance utilizes the full training scheme without resetting the input qubits.

NOTE: Because QCompress was designed to run on the quantum device (as well as the simulator), we need to anticipate nontrival mappings between abstract qubits and physical qubits. The dictionaries `q_in`, `q_latent`, and `q_refresh` are abstract-to-physical qubit mappings for the input, latent space, and refresh qubits respectively. A cool plug-in/feature to add would be to have an automated "qubit mapper" to determine the optimal or near-optimal abstract-to-physical qubit mappings for a particular QAE instance.

[2]:
```
### QAE setup options

# Abstract-to-physical qubit mapping
q_in = {'q0': 0, 'q1': 1} # Input qubits
q_latent = {'q1': 1}      # Latent space qubits
q_refresh = {'q2': 2}     # Refresh qubits

# Training scheme setup: Full without reset feature
trash_training = False
reset = False

# Simulator settings
cxn_setting = '3q-qvm'
n_shots = 5000
```

2.5.2 Data preparation circuits

To prepare the quantum data, we define the state preparation circuits (and their daggered circuits). In this particular example, we will generate the data by scanning over various values of `phi`.

[3]:
```
def _state_prep_circuit(phi, qubit_indices):
    """
    Returns parametrized state preparation circuit.
    We will vary over phi to generate the data set.

    :param phi: (list or numpy.array, required) List or array of data generation␣
↪parameters
    :param qubit_indices: (list, required) List of qubit indices
    :returns: State preparation circuit
    :rtype: pyquil.quil.Program
    """
    circuit = Program()
    circuit += Program(RY(phi[0], qubit_indices[1]))
```

(continues on next page)

(continued from previous page)

```
    circuit += Program(CNOT(qubit_indices[1], qubit_indices[0]))
    return circuit

def _state_prep_circuit_dag(phi, qubit_indices):
    """
    Returns the daggered version of the state preparation circuit.

    :param phi: (list or numpy.array, required) List or array of data generation␣
↪parameters
    :param qubit_indices: (list, required) List of qubit indices
    :returns: State un-preparation circuit
    :rtype: pyquil.quil.Program
    """
    circuit = Program()
    circuit += Program(CNOT(qubit_indices[1], qubit_indices[0]))
    circuit += Program(RY(-phi[0], qubit_indices[1]))
    return circuit
```

2.5.3 Qubit labeling

In the cell below, we produce lists of **ordered** physical qubit indices involved in the compression and recovery maps of the quantum autoencoder. Depending on the training and reset schemes, we may use different qubits for the compression vs. recovery.

[4]:
```
compression_indices = order_qubit_labels(q_in).tolist()

q_out = merge_two_dicts(q_latent, q_refresh)
recovery_indices = order_qubit_labels(q_out).tolist()

if not reset:
    recovery_indices = recovery_indices[::-1]

print("Physical qubit indices for compression : {0}".format(compression_indices))
print("Physical qubit indices for recovery    : {0}".format(recovery_indices))
```

```
Physical qubit indices for compression : [0, 1]
Physical qubit indices for recovery    : [2, 1]
```

For the full training scheme with no resetting feature, this will require the three total qubits.

The first two qubits (`q0`, `q1`) will be used to encode the quantum data. `q1` will then be used as the latent space qubit, meaning our objective will be to reward the training conditions that "push" the information to the latent space qubit. Then, a refresh qubit, `q2`, is added to recover the original data.

2.5.4 Data generation

After determining the qubit mapping, we add this physical qubit information to the state preparation circuits and store the "mapped" circuits.

[5]:
```
# Lists to store state preparation circuits
list_SP_circuits = []
list_SP_circuits_dag = []

phi_list = np.linspace(-pi/2., pi/2., 40)
```

(continues on next page)

(continued from previous page)

```
for angle in phi_list:

    # Map state prep circuits
    state_prep_circuit = _state_prep_circuit([angle], compression_indices)

    # Map daggered state prep circuits
    if reset:
        state_prep_circuit_dag = _state_prep_circuit_dag([angle], compression_indices)
    else:
        state_prep_circuit_dag = _state_prep_circuit_dag([angle], recovery_indices)

    # Store mapped circuits
    list_SP_circuits.append(state_prep_circuit)
    list_SP_circuits_dag.append(state_prep_circuit_dag)
```

2.5.5 Training circuit preparation

In this step, we choose a parametrized quantum circuit that will be trained to compress then recover the input data set.

NOTE: This is a simple one-parameter training circuit.

[6]:
```
def _training_circuit(theta, qubit_indices):
    """
    Returns parametrized/training circuit.

    :param theta: (list or numpy.array, required) Vector of training parameters
    :param qubit_indices: (list, required) List of qubit indices
    :returns: Training circuit
    :rtype: pyquil.quil.Program
    """
    circuit = Program()
    circuit += Program(RY(-theta[0]/2, qubit_indices[0]))
    circuit += Program(CNOT(qubit_indices[1], qubit_indices[0]))
    return circuit

def _training_circuit_dag(theta, qubit_indices):
    """
    Returns the daggered parametrized/training circuit.

    :param theta: (list or numpy.array, required) Vector of training parameters
    :param qubit_indices: (list, required) List of qubit indices
    :returns: Daggered training circuit
    :rtype: pyquil.quil.Program
    """
    circuit = Program()
    circuit += Program(CNOT(qubit_indices[1], qubit_indices[0]))
    circuit += Program(RY(theta[0]/2, qubit_indices[0]))
    return circuit
```

As was done for the state preparation circuits, we also map the training circuits with physical qubits we want to use.

[7]:
```
training_circuit = lambda param : _training_circuit(param, compression_indices)

if reset:
```

(continues on next page)

(continued from previous page)

```
    training_circuit_dag = lambda param : _training_circuit_dag(param, compression_
↪indices)
else:
    training_circuit_dag = lambda param : _training_circuit_dag(param, recovery_
↪indices)
```

2.5.6 Define the QAE instance

Here, we initialize a QAE instance. This is where the user can decide which optimizer to use, etc. For this demo, we use the default COBYLA optimizer.

[8]:
```
qae = quantum_autoencoder(state_prep_circuits=list_SP_circuits,
                          training_circuit=training_circuit,
                          q_in=q_in,
                          q_latent=q_latent,
                          q_refresh=q_refresh,
                          state_prep_circuits_dag=list_SP_circuits_dag,
                          training_circuit_dag=training_circuit_dag,
                          trash_training=trash_training,
                          reset=reset,
                          n_shots=n_shots,
                          print_interval=1)
```

After defining the instance, we set up the Forest connection (in this case, a simulator) and split the data set.

[9]:
```
qae.setup_forest_cxn(cxn_setting)
```

[10]:
```
qae.train_test_split(train_indices=[1, 31, 16, 7, 20, 23, 9, 17])
```

[11]:
```
print(qae)
```

```
QCompress Setting
=================
QAE type: 2-1-2
Data size: 40
Training set size: 8
Training mode: full cost function
  Reset qubits: False
Compile program: False
Forest connection: 3q-qvm
  Connection type: QVM
```

2.5.7 Loss landscape generation and visualization

For small enough examples, we can visualize the loss landscape, which can help us understand where the minimum is. This might be more useful when simulating a noisy version of the autoencoder.

[12]:
```
# Collect loss landscape data (scan over various values of theta)
theta_scan = np.linspace(-pi, pi, 30)
training_losses = []

for angle in theta_scan:
    print("Theta scan: {}".format(angle))
```

(continues on next page)

(continued from previous page)

```
    angle = [angle]
    training_loss = qae.compute_loss_function(angle)
    training_losses.append(training_loss)
```

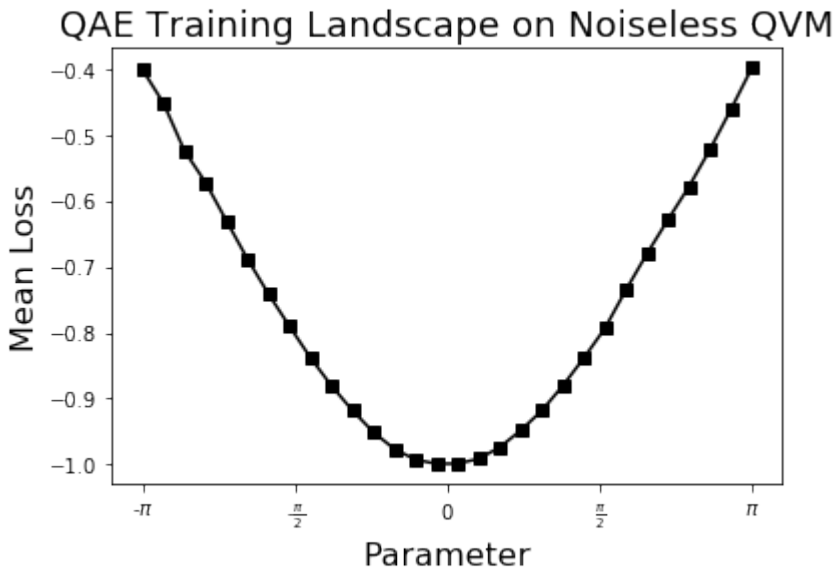
```
Theta scan: -3.141592653589793
Iter    0 Mean Loss: -0.4013250
Theta scan: -2.9249310912732556
Iter    1 Mean Loss: -0.4522250
Theta scan: -2.708269528956718
Iter    2 Mean Loss: -0.5245500
Theta scan: -2.4916079666401805
Iter    3 Mean Loss: -0.5723250
Theta scan: -2.2749464043236434
Iter    4 Mean Loss: -0.6308250
Theta scan: -2.058284842007106
Iter    5 Mean Loss: -0.6885500
Theta scan: -1.8416232796905683
Iter    6 Mean Loss: -0.7416250
Theta scan: -1.624961717374031
Iter    7 Mean Loss: -0.7906250
Theta scan: -1.4083001550574934
Iter    8 Mean Loss: -0.8383000
Theta scan: -1.1916385927409558
Iter    9 Mean Loss: -0.8808250
Theta scan: -0.9749770304244185
Iter   10 Mean Loss: -0.9182750
Theta scan: -0.758315468107881
Iter   11 Mean Loss: -0.9512250
Theta scan: -0.5416539057913434
Iter   12 Mean Loss: -0.9767250
Theta scan: -0.3249923434748059
Iter   13 Mean Loss: -0.9926000
Theta scan: -0.10833078115826877
Iter   14 Mean Loss: -0.9987500
Theta scan: 0.10833078115826877
Iter   15 Mean Loss: -0.9985500
Theta scan: 0.3249923434748063
Iter   16 Mean Loss: -0.9911500
Theta scan: 0.5416539057913439
Iter   17 Mean Loss: -0.9738000
Theta scan: 0.7583154681078814
Iter   18 Mean Loss: -0.9480500
Theta scan: 0.9749770304244185
Iter   19 Mean Loss: -0.9182250
Theta scan: 1.191638592740956
Iter   20 Mean Loss: -0.8804500
Theta scan: 1.4083001550574936
Iter   21 Mean Loss: -0.8384500
Theta scan: 1.6249617173740312
Iter   22 Mean Loss: -0.7935500
Theta scan: 1.8416232796905687
Iter   23 Mean Loss: -0.7355000
Theta scan: 2.0582848420071063
Iter   24 Mean Loss: -0.6805250
Theta scan: 2.274946404323644
Iter   25 Mean Loss: -0.6289750
Theta scan: 2.4916079666401814
```

(continues on next page)

(continued from previous page)

```
Iter   26 Mean Loss: -0.5787250
Theta scan: 2.708269528956718
Iter   27 Mean Loss: -0.5225000
Theta scan: 2.9249310912732556
Iter   28 Mean Loss: -0.4615500
Theta scan: 3.141592653589793
Iter   29 Mean Loss: -0.3984500
```

[15]:
```
# Visualize loss landscape
fig = plt.figure(figsize=(6, 4))
plt.plot(theta_scan, np.array(training_losses), 'ks-')
plt.title("QAE Training Landscape on Noiseless QVM", fontsize=18)
plt.xlabel("Parameter", fontsize=16)
plt.ylabel("Mean Loss", fontsize=16)
plt.xticks([-np.pi, -np.pi/2., 0., np.pi/2., np.pi],
           [r"-$\pi$", r"-$\frac{\pi}{2}$", "$0$", r"$\frac{\pi}{2}$", r"$\pi$"])
plt.show()
```



[ ]:

2.5.8 A slightly larger example

We've tried doing a similar landscape scan for the hydrogen example shown in qae_h2_demo.ipynb.

In this hydrogen example, we used a two-parameter training circuit.

This is the loss landscape for the full training case with no reset feature on the noiseless simulator. The number of circuit shots used is 3000. We can see that the minimum is at (,).

[ ]:

CHAPTER 3

API Reference

3.1 qcompress

3.1.1 qcompress.qae_engine module

3.1.2 qcompress.utils module

CHAPTER 4

Indices and tables

- genindex
- modindex
- search