Parameterised quantum circuits for supervised learning

Qing Zhou, s2501597

January 8, 2021

Abstract

In this project, we used parametrised quantum circuits (PQCs) for supervised learning, in order to solve a classification problem. Different aspects of the impact of the model parameters on the model performance was investigated, with a particular focus on the circuit architecture, number of qubits, number of layers and optimisation method. It was found for our particular chosen problem of classifying the hand-written digits, PQCs can achieve remarkable results, even with very limited dimensionality of data and training size. It can be anticipated, PQCs might be very useful for tacking datasets of this kind, and in some cases, even outperform classical neural networks (e.g. Broughton et al., 2020).

1 Introduction

Very much like a classical computer that uses bits and logic gates when processing binary operations, in the realm of quantum computing, the quantum equivalent of bits and logic gates – namely "qubits" and "quantum logic gates" are utilised, complemented by superposition and entanglement supremacies. There is a wide variety of quantum algorithms developed to tackle all kinds of problems, one of which is variational hybrid quantum classical (HQC) algorithm, implementing both quantum and classical computers to accomplish a particular task. With the development of Noisy Intermediate-Scale Quantum (NISQ) technology, quantum devices will be able to support 50 -100 qubits and around 1000 gate operations (Sim et al., 2019), making HQC algorithm all the more important.

Parametrised quantum circuits (PQCs) are the common framework in implementations of HQC algorithms such as variational quantum eigensolver (VQE) and the quantum approximate optimisation algorithm (QAOA). PQCs often involve a series of gates, controlled gates such as controlled NOT gates and adjustable gates such as rotation gates. In HQC algorithms, the PQCs work as the interface between the quantum block and the classical block, where the former takes quantum data as inputs, and the latter tunes the parameters coming out of the quantum block. Based on a PQC we can build a supervised learning model. Fig. 1 illustrates a supervised learning model using HQC algorithm.

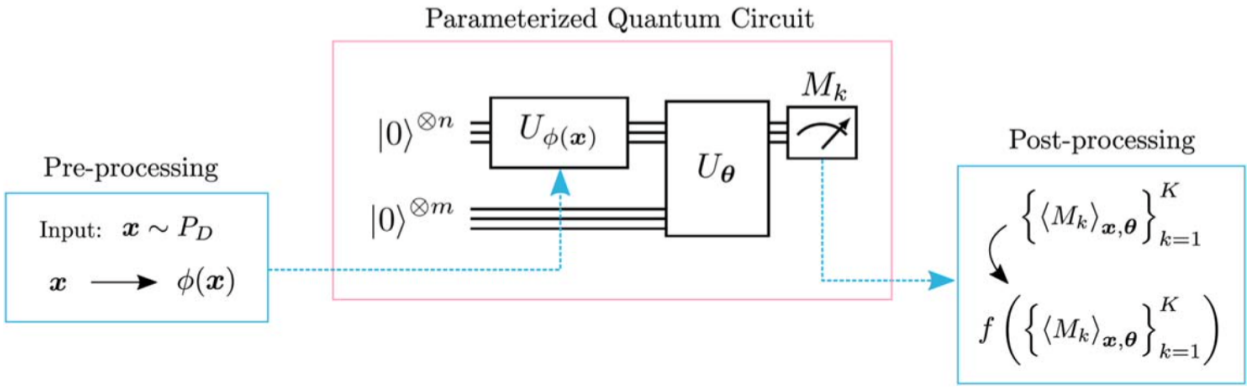


Figure 1: A supervised learning model implementing HQC algorithm. The model consists of three components: the classical pre/post processing blocks and the quantum block with PQC. Adapted from Fig. 2 by Benedetti et al. (2019).

In the pre-processing step, the classical data is embedded into quantum states. One way of doing this is by mapping the vector components of an input data into amplitudes of a quantum state. In addition, pre-processing methods such as standardising or PCA also applies. The transformed data will subsequently be fed into the

circuit with a sequence of gates, and the outcome will then be used to compute the loss.
Before getting into the details of the formalising of the objectives, there is one thing we want to know, that is, the "effectiveness" of a PQC. For a specific task to solve, how do we decide which circuit architecture is best suited for our problem of interest? Remember that our goal is to maximise the generalisation ability of a supervised task, in a cost-effective way. This question can be answered quantitatively with the help of circuit descriptors, for instance expressibility or entangling capacity.

1.1 Descriptors of PQCs

1.1.1 Expressibility

Expressibility can be understood as a circuit's ability to generate states from the Hilbert space. One way of computing it is by measuring the deviations of the states generated from the PQC to the uniform distribution in the state space. The measure of expressibility can be computed by taking the Kullback-Leibner divergence (KL diver- gence) between the estimated fidelity distribution and that of the Haar distributed ensemble (Sim et al., 2019).

$$\mathrm{Expr} = \mathrm{D_{KL}}(\hat{\mathrm{P}}_{\mathrm{PQC}}(\mathrm{F};\theta) \parallel \mathrm{P_{Haar}}(\mathrm{F})) \tag{1}$$

where $\hat{P}_{\mathrm{PQC}}(F;\theta)$ is the estimated probability distribution of fidelities resulting from sampling states from a PQC. It can be seen, that expressibility is also the amount of information lost when we were to approximate the distribution of fidelities from a PQC to the Haar distribution.

1.1.2 Entangling capacity

Entangling capacity is a way of quantifying the degree of entanglement of a PQC. It is quantified by the Meyer-Wallach measure as (Sim et al., 2019)

$$\mathrm{Ent} = \frac{1}{|\mathrm{S}|} \sum_{\theta_\mathrm{i} \in \mathrm{S}} \mathrm{Q}(|\psi_{\theta_\mathrm{i}}\rangle) \tag{2}$$

where $S = \{\theta_i\}$ is the set of sampled circuit parameter vectors. For highly entangled states such as Bell states, the score will be close to 1.

1.1.3 Circuit costs

Other descriptors quantifying the cost of a PQC also exist, such as circuit depth, and number of parameters. While our goal is to design the circuit structure in a way to achieve the best possible performance on the test set, we cannot do this with an infinite cost. We must find a balance between the circuit cost and model efficiency.

1.2 Outline

The structure of this report is as follows: in section 2 we introduce the design of the supervised learning model, including the choosing of the dataset, the circuit templates that will be used in the following experiments, accompanied by the descriptors. A typical model of the circuit will be explained, as well as the steps for the post-processing part. In section 3 the results of different experiments will be presented, as well as their implications. In section 4, a summary of this report will be delivered, as well as implications for limitations and future works.

2 Methods

2.1 Dataset and data pre-processing

In all following experiments discussed, the dataset used were chosen to be the MNIST dataset (LeCun et al., 2010), distinguishing digits 3 and 6. The original data consists of images of pixel size 28×28, which is too large for the quantum circuit to process. Using `decomposition.TruncatedSVD` from the `scikit-learn` library (Buitinck et al., 2013) which implements latent semantic analysis (LSA), the dimensionality of the data were able to be reduced to the number of qubits of the quantum circuit of interest. The features of the data were subsequently standardised and scaled to unit variance with `sklearn.preprocessing.StandardScaler`. In order to improve computational cost, a subset of the original data was made, so that only 240 data points were kept, half of which were labelled as "3"s. Fig. 2.

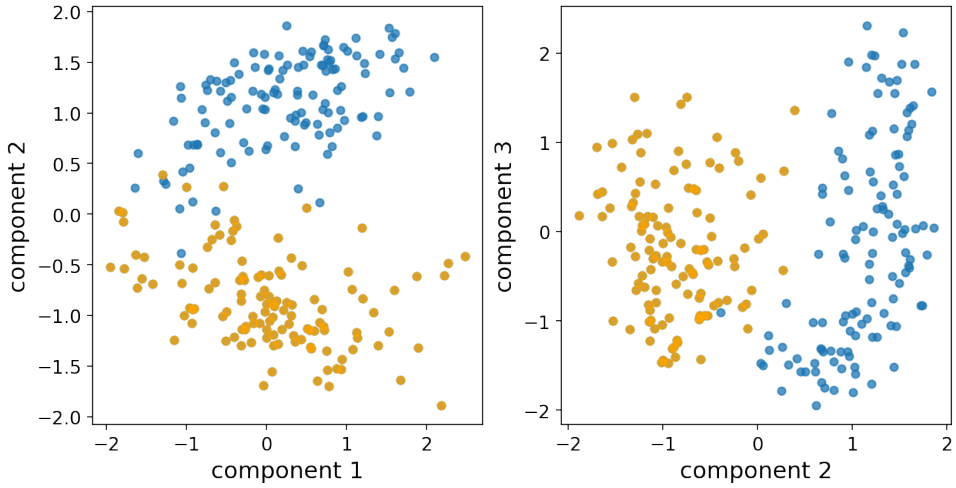


Figure 2: Scatter plot of the subsample used for all experiments, after dimensionality reduction with LSA algorithm. The orange markers represent "3"s and number of dimension shown here is 3. *Left:* scatter plot projected on component 1 and component 2; *Right:* scatter plot projected on component 2 and component 3.

The motivation for this data pre-procession was that, we want to model the effect of a real quantum device using a traditional computer, and at the same time we want our data to still provide sufficient information for the model to distinguish one class from another. From Fig. 2 it can be seen, even when reduced to only 3 dimensions, there is still a clear distinction between the two classes by eye, and with proper circuit architecture this distinction should be captured by the model as well.

2.2 Circuit templates and descriptors

Different templates were chosen in order to investigate the relation between circuit architecture and model accuracy. Motivated by Sim et al. (2019), the following circuit templates presented in Fig. 3 were selected. Note that only templates with 3 qubits were presented, for 2 qubits and 4 qubits, the circuit architecture varies with gates parameters listed in Table 1.

Two descriptors were used in order to gauge the cost of the selected templates: number of parameters and number of two-qubit gates. The corresponding parameter values can be found in Table 1.

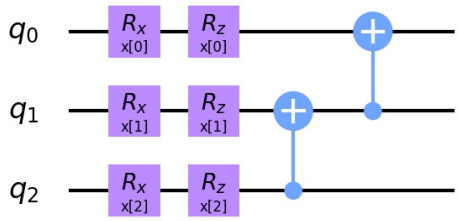
Table 1: Cost parameters for the 6 circuit templates, where n denotes the number of qubits, L represents the number of layers

Template ID	Number of parameters	Number of two-qubit gates
Template 1	$2nL$	$(n-1)L$
Template 2	$2nL$	$(n-1)L$
Template 3	$2(3n-1)L$	$(n-1)L$
Template 4	$2nL$	$(n-1)L$
Template 5	$(2n+2)L$	$(n-1)L$
Template 6	$2nL$	$2nL$

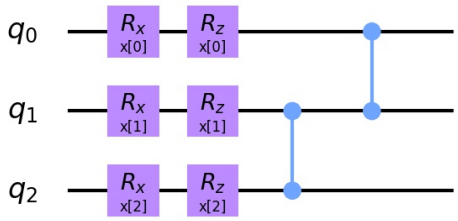
2.3 Variational quantum classifier

In order to solve the problem of classifying hand-written digits of 3 and 6, we implemented variational quantum classification algorithm with `aqua.algorithms.VQC` from Qiskit (Abraham et al., 2019). The basic routine for each run is as follows:

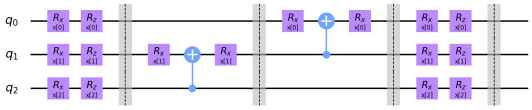
1. Select a circuit template from the available 6 templates as the feature map object for the VQC object;
2. Select an variational circuit for the VQC object, which works as a benchmark to compute the cost function (for all experiments presented, the variational circuit was chosen to be the `TwoLocal` circuit with number of qubits the same as the feature map's);
3. Select an optimiser for the VQC object;



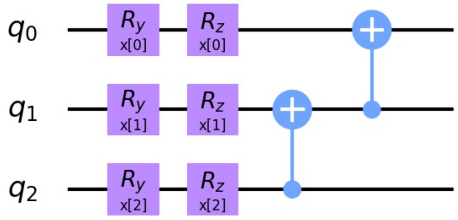
(a) circuit template 1.



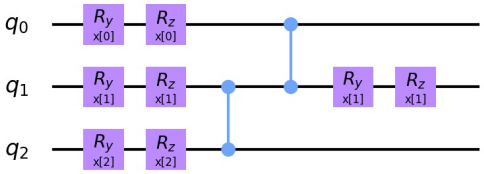
(b) circuit template 2.



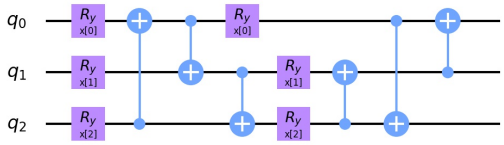
(c) circuit template 3.



(d) circuit template 4.



(e) circuit template 5.



(f) circuit template 6.

Figure 3: A set of 3-qubit circuit templates used in this project (note that most experiments were carried out with 3 qubit in order to lower the computational cost, but when investigating the effect of changing the number of qubits, the templates change accordingly).

4. Vary the number of qubits and/or the number of circuit layer when necessary.

For each run, a 5-fold cross-validation was implemented, and an average of the accuracies on the test set of the 5 folds was computed and stored for further discussion.

2.4 Parameter tuning

Even with the simplest circuit template a complete run took about 200 seconds. Since the goal of our project is to investigate the impact of the parameters on the model performance, instead of using exhaustive grid search algorithm for the best suite of parameters, the following was done:

1. For each circuit template, change only the number of layers to be 1,2, and 3 and calculate the average accuracy on the 5 folds;
2. Change only the optimiser for a particular chosen circuit template (template 4) and gauge the impact of optimisation method on the performance of the model;
3. Change only the number of qubits for a particular chosen circuit template (template 4) and see how the performance of the model improve/reduce.

3 Experiments

3.1 A trade-off between circuit cost and model efficiency

Fig. 4 presents the results of model accuracy on the test set with 5-fold cross-validation, when varying circuit templates and number of layers at the same time. Here by default the optimiser used was COBYLA (Constrained

Optimization By Linear Approximation), with 3 qubits.

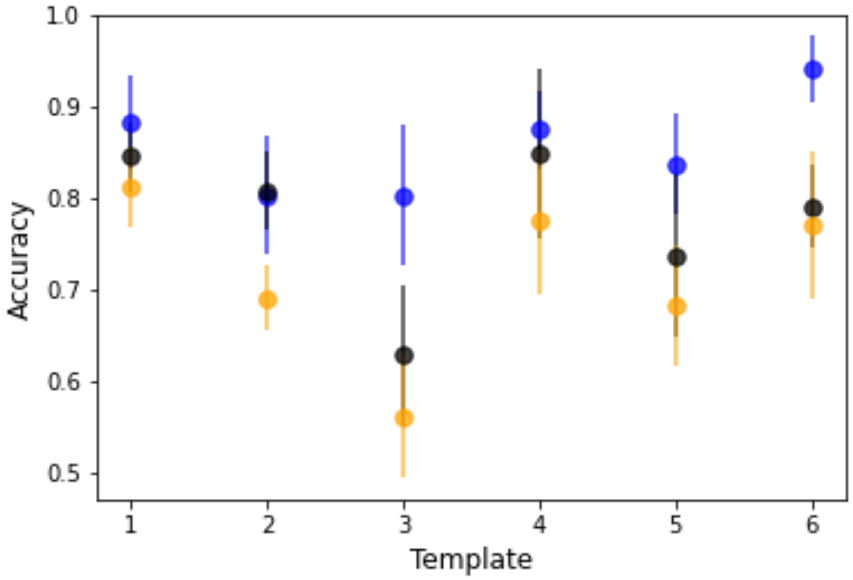


Figure 4: Model accuracy on the test set with 5-fold cross-validation, for different circuit templates and number of circuit layers.

In overall, there is a trend of decreasing model accuracy when increase the number of layers. Sim et al. (2019) found that there is a strong correlation between circuit expressibility and the number of circuit layers, such that the expressibility normally increase with increasing number of layers. Hubregtsen et al. (2020) found that expressibility also relates to the model accuracy and for most of the datasets higher accuracy corresponds to also higher expressibility. The trend found in this experiment might be due to the fact that, for dataset with small dimensions and of small size, a more suitable choice of the quantum circuit used for classification might be the one with less fewer number of parameters but more number of two-qubit gates per layer, i.e. the expressibility should be lower so that the ability of the circuit to generate random states that are representative of the Bloch sphere is lower, for the model to deliver higher accuracy.
This result suggests that, even with the simplest circuit architecture and limited number of qubits, PQCs are able to achieve relatively stable training results (e.g. template 6 and template 4), when tacking a particular sort of questions.

3.2 The effect of optimiser

The impact of optimisation method on the model performance is somewhat harder to gauge. First of all, all optimisers available has there own parameters (e.g. for the Adam optimiser the user should specify the learning rate whereas for others there is no such parameter), an exhaustive number of experiments are needed to decide the optimal set of parameters for each parameter. Secondly, it is not necessarily the case that the number of folds (default to be 5 in all experiments) used for cross-validation is the best one for a particular optimiser, for instance in the case of the Adam optimiser, with 2 fold cross-validation the result was better than that with 5 folds. Here in our report with limited time and limited computation resource, we only used one set of parameters for each optimiser, with no guarantee that the chosen set works the best. Fig. 5 presents a very crude result of the accuracy change with varying optimisation methods, template 4 was used for all runs in the figure. With COBYLA, the accuracy is the steadiest and with the highest mean, it is also chosen as the default optimisation method in the following experiments.

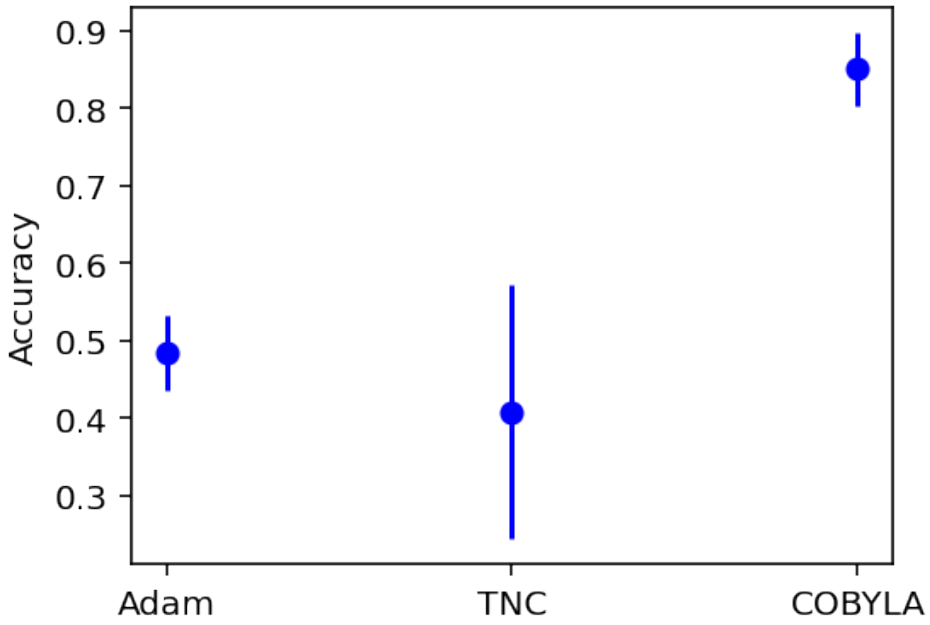


Figure 5: The effect of changing the optimisers used on the model performance

3.3 The effect of increasing the number of qubits

Very often, in a supervised learning task, when the performance of the model does not get any better with more epochs, one seeks a way to improve the complexity of the network in order to break the convergence issue. It is interesting to find out whether or not increasing the number of qubits has a positive impact on our model accuracy. Fig. 6 presents the result of varying the number of qubits used from 3 to 4 and 5.

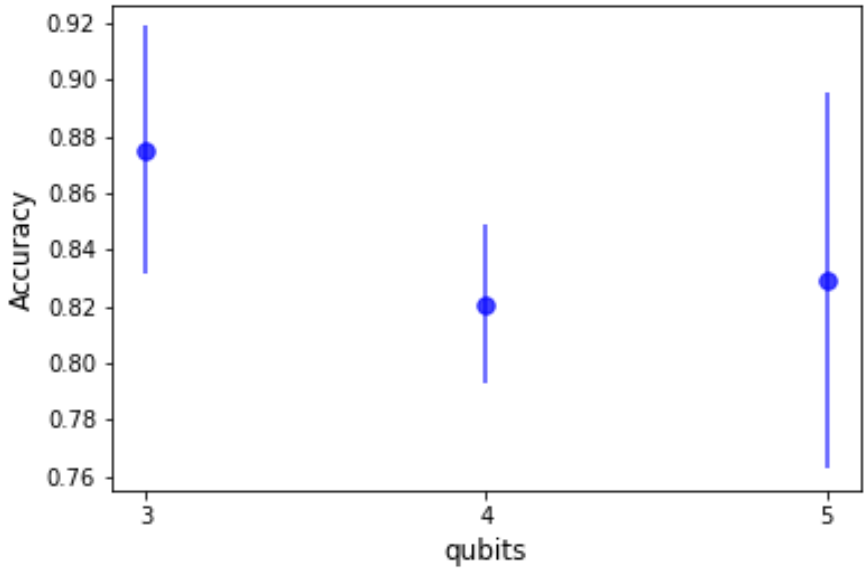


Figure 6: The effect of changing the number of qubits used on the model performance.

With 3 qubits, the accuracy is the highest, albeit not the steadiest, and with 5 qubits, the scatter increased greatly and with smaller accuracy. This could be explained in two folds: the chosen dataset favours circuit with lower complexity and lower expressibility; the steadiness of the circuit decrease with more qubits.

4 Conclusion

In this project, we implemented PQC for supervised learning of a classification problem. The impacts of various parameters of the model on the training accuracy was investigated, with a particular focus on the circuit architecture, number of layers, optimisation method and number of qubits. With limited time and resource, the goal was not to find the optimal suite of parameters for this particular problem, instead, it was to investigate how each parameter impact the performance of the model, and make some educated guess for the causes. The eye-catching result was that, without increasing the complexity of the circuit architecture (number of layers, number of qubits) greatly, we can already achieve steady and pleasant model accuracy. Results from section 3.1 and section 3.3 showed that, changing circuit architecture might be a good way to improve the model performance cost-effectively. It might even deliver faster and better results when compared to increasing the complexity of the circuit.
Another aspect might also be explored for this work, was to gauge the impact of input datasets on the model performance. With more available datasets it would be better to see which kind of problem was best suited for a simple PQC with one layer and 3 qubits to solve, so that we might not have to increase the complexity of the PQC when solving some specific problems.

References

Abraham, H., AduOffei, Agarwal, R., et al. 2019, Qiskit: An Open-source Framework for Quantum Computing, doi:10.5281/zenodo.2562110

Benedetti, M., Lloyd, E., Sack, S., & Fiorentini, M. 2019, Quantum Science and Technology, 4, 043001

Broughton, M., Verdon, G., McCourt, T., et al. 2020, arXiv preprint arXiv:2003.02989

Buitinck, L., Louppe, G., Blondel, M., et al. 2013, in ECML PKDD Workshop: Languages for Data Mining and Machine Learning, 108–122

Hubregtsen, T., Pichlmeier, J., & Bertels, K. 2020, arXiv preprint arXiv:2003.09887

LeCun, Y., Cortes, C., & Burges, C. 2010, ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist, 2

Sim, S., Johnson, P. D., & Aspuru-Guzik, A. 2019, Advanced Quantum Technologies, 2, 1900070

Appendix

```python
import numpy as np
import tensorflow as tf
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import StandardScaler


#load mnist data
mnist = tf.keras.datasets.mnist.load_data()
(x_train, y_train), (x_test, y_test) = mnist

def filter_36(x, y):
    "function that filters 3 and 6 of the mnist dataset"
    keep = (y == 3) | (y == 6)
    x, y = x[keep], y[keep]

    return x,y
# get filtered data
x_train, y_train = filter_36(x_train, y_train)
x_test, y_test = filter_36(x_test, y_test)
# stack training set and test set
# create features  and labels
features_36 = np.vstack((x_train,x_test))
labels_36 = np.concatenate((y_train,y_test))
tot_num = len(features_36)
# flatten features
features_36 = features_36.reshape(tot_num,784)

```

```
28
29 # use truncated SVD for dimensionality reduction
30 # to datasets of dimension 3,4, and 5
31 svd = TruncatedSVD(n_components=3, n_iter=5, random_state=42)
32 features_embedded3 = svd.fit_transform(features_36)
33
34 svd = TruncatedSVD(n_components=4, n_iter=5, random_state=42)
35 features_embedded4 = svd.fit_transform(features_36)
36
37 svd = TruncatedSVD(n_components=5, n_iter=5, random_state=42)
38 features_embedded5 = svd.fit_transform(features_36)
39
40 # standard scaling the dataset
41 scaler = StandardScaler()
42 features_embedded3 = scaler.fit_transform(features_embedded3)
43 features_embedded4 = scaler.fit_transform(features_embedded4)
44 features_embedded5 = scaler.fit_transform(features_embedded5)
45
46 # stack labels and transformed features and write new data to file
47 data1 = np.column_stack((labels_36,features_embedded3))
48 data2 = np.column_stack((labels_36,features_embedded4))
49 data3 = np.column_stack((labels_36,features_embedded5))
50
51 data1.tofile('data1.txt')
52 data2.tofile('data2.txt')
53 data3.tofile('data3.txt')
```

data_prep.py

```
1  import time
2  import numpy as np
3  from qiskit import *
4
5  from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
6  from qiskit import Aer
7  from qiskit.aqua.utils import split_dataset_to_data_and_labels, map_label_to_class_name
8  from qiskit.aqua import QuantumInstance
9  from qiskit.aqua.algorithms import VQC
10 from qiskit.circuit import ParameterVector
11 from qiskit.circuit.library import  TwoLocal, NLocal
12 from qiskit.aqua.components.optimizers import ADAM, SPSA, COBYLA
13
14
15 dat1 = np.fromfile('data1.txt').reshape(-1,4)
16 ind  = np.concatenate((np.where(dat1[:,0]==3),np.where(dat1[:,0]!=3)),axis=None)
17 data1 = dat1[ind]
18 data1 = np.vstack((data1[:120],data1[-120:]))
19
20 dat2 = np.fromfile('data2.txt').reshape(-1,5)
21 ind  = np.concatenate((np.where(dat2[:,0]==3),np.where(dat2[:,0]!=3)),axis=None)
22 data2 = dat2[ind]
23 data2 = np.vstack((data2[:120],data2[-120:]))
24
25 dat3 = np.fromfile('data3.txt').reshape(-1,6)
26 ind  = np.concatenate((np.where(dat3[:,0]==3),np.where(dat3[:,0]!=3)),axis=None)
27 data3 = dat3[ind]
28 data3 = np.vstack((data3[:120],data3[-120:]))
29
30 class DataSet(object):
31
32     """Docstring for CrossValidation. """
33
34     def __init__(self, data, k=None, A=3, B=6):
35         """TODO: to be defined. """
36         self.data = data.copy()
37         self.k = k
38         self.A = A
39         self.B = B
40         self.dataA = self.data[self.data[:,0]==self.A]
41         self.dataB = self.data[self.data[:,0]==self.B]
42
43     def CV(self, data):
44         k = self.k
45         size = len(data)
46         index = np.arange(size)
```

```
47          np.random.shuffle(index)
48          DATA_IND={}
49          ind_split = np.array_split(index, k)
50          for name in range(k):
51              sub = {}
52              train_split = []
53              for i in range(k):
54                  if i==name:
55                      sub['test_ind'] = ind_split[i]
56                  else:
57                      train_split.append(ind_split[i])
58              sub['train_ind'] = np.hstack(train_split)
59              DATA_IND.update({name: sub})
60          return DATA_IND, index
61
62      def data_gen(self):
63          ind, _ = self.CV(self.dataA)
64          for i in range(self.k):
65              tr_input = {
66                  'A': self.dataA[ind[i]['train_ind']][:,1:],
67                  'B': self.dataB[ind[i]['train_ind']][:,1:]}
68              te_input = {
69                  'A': self.dataA[ind[i]['test_ind']][:,1:],
70                  'B': self.dataB[ind[i]['test_ind']][:,1:]}
71              te_label =
    (self.dataA[ind[i]['test_ind']][:,:1],self.dataB[ind[i]['test_ind']][:,:1])
72              yield {
73                  'k': i,
74                  'tr_input': tr_input,
75                  'te_input': te_input,
76                  'testing_input': np.concatenate((te_input['A'], te_input['B'])),
77                  'testing_label': np.concatenate(te_label)
78                    }
79
80
81
82  class Main(object):
83
84      """Docstring for CrossValidation. """
85
86      def __init__(self, reps=2, num_qubits=3, fm_func=None, vc_func=None, seed=10598):
87          """TODO: to be defined. """
88          self.reps = reps
89          self.num_qubits = num_qubits
90          self.fm_func = fm_func(reps=self.reps, num_qubits=self.num_qubits)
91          self.vc_func = vc_func(num_qubits=self.num_qubits)
92          self.seed = seed
93          self.backend = Aer.get_backend('qasm_simulator')
94          self.backend_options = {"method": "statevector", "max_parallel_threads": 0,
    "max_parallel_experiments": 4,"max_parallel_shots": 1}
95
96      def drawfmap(self, fname):
97          return self.fm_func.draw(output='mpl', filename=fname+'_fmap.jpg')
98
99      def drawvmap(self, fname):
100         return self.vc_func.draw(output='mpl', filename=fname+'_var_circ.jpg')
101
102     def call_back_vqc(self, eval_count, var_params, eval_val, index):
103         text = "index({}): current cross entropy cost: {}".format(eval_count, eval_val)
104         #print(text)
105
106     def optimization(self, training_input, test_input):
107         self.quantum_instance = QuantumInstance(
108             self.backend,
109             shots=1024,
110             seed_simulator=self.seed,
111             seed_transpiler=self.seed,
112             backend_options=self.backend_options)
113         self.vqc = VQC(optimizer=self.opt,
114           feature_map=self.fm_func,
115           var_form=self.vc_func,
116           callback=self.call_back_vqc,
117           training_dataset=training_input,
118           test_dataset=test_input)
119
```

```
120     def train(self, training_input, test_input):
121         self.optimization(training_input, test_input)
122         start = time.process_time()
123 
124         result = self.vqc.run(self.quantum_instance)
125 
126         print("time taken: ")
127         print(time.process_time() - start)
128         print("testing success ratio: {}".format(result['testing_accuracy']))
129         return result['testing_accuracy']
130 
131 
132 
133 
134 def feature_map_expr1(reps=2, num_qubits=3):
135     feature_map = QuantumCircuit(num_qubits)
136     x = ParameterVector('x', length=num_qubits)
137 
138     #EXPERIMENT 3: 72.9
139     for _ in range(reps):
140         for i in range(num_qubits):
141             feature_map.rx(x[i], i)
142             feature_map.rz(x[i], i)
143         for i in range(num_qubits-1, 0, -1):
144             feature_map.cx(i, i-1)
145     return feature_map
146 
147 
148 def feature_map_expr2(reps=2, num_qubits=3):
149     feature_map = QuantumCircuit(num_qubits)
150     x = ParameterVector('x', length=num_qubits)
151 
152     #EXPERIMENT 4: 71.2
153     for _ in range(reps):
154         for i in range(num_qubits):
155             feature_map.rx(x[i], i)
156             feature_map.rz(x[i], i)
157         for i in range(num_qubits-1, 0, -1):
158             feature_map.cz(i, i-1)
159     return feature_map
160 
161 
162 
163 def feature_map_expr3(reps=2, num_qubits=3):
164     feature_map = QuantumCircuit(num_qubits)
165     x = ParameterVector('x', length=num_qubits)
166 
167 
168     for _ in range(reps):
169         for i in range(num_qubits):
170             feature_map.rx(x[i], i)
171             feature_map.rz(x[i], i)
172         feature_map.barrier()
173         for control in range(num_qubits-1, 0, -1):
174             target = control - 1
175             feature_map.rx(x[target], target)
176             feature_map.cx(control, target)
177             feature_map.rx(x[target], target)
178             feature_map.barrier()
179         for i in range(num_qubits):
180             feature_map.rx(x[i], i)
181             feature_map.rz(x[i], i)
182         feature_map.barrier()
183     return feature_map
184 
185 def feature_map_expr4(reps=2, num_qubits=3):
186 
187     feature_map = QuantumCircuit(num_qubits)
188     x = ParameterVector('x', length=num_qubits)
189 
190     # Experiment 13: 70.1
191     for _ in range(reps):
192         for i in range(num_qubits):
193             feature_map.ry(x[i], i)
194             feature_map.rz(x[i], i)
```

```
195         for i in range(num_qubits - 1, 0, -1):
196             feature_map.cx(i, i-1)
197 
198     return feature_map
199 
200 def feature_map_expr5(reps=2, num_qubits=3):
201     feature_map = QuantumCircuit(num_qubits)
202     x = ParameterVector('x', length=num_qubits)
203 
204     for _ in range(reps):
205         for i in range(num_qubits):
206             feature_map.ry(x[i], i)
207             feature_map.rz(x[i], i)
208         for i in range(num_qubits - 1, 0, -1):
209             feature_map.cz(i, i-1)
210         feature_map.ry(x[1], 1)
211         feature_map.rz(x[1], 1)
212     return feature_map
213 
214 
215 def feature_map_expr6(reps=2, num_qubits=3):
216     feature_map = QuantumCircuit(num_qubits)
217     x = ParameterVector('x', length=num_qubits)
218 
219     for _ in range(reps):
220         for i in range(num_qubits):
221             feature_map.ry(x[i], i)
222         feature_map.cx(num_qubits-1, 0)
223         for i in range(num_qubits-1):
224             feature_map.cx(i, i+1)
225         for i in range(num_qubits):
226             feature_map.ry(x[i], i)
227         feature_map.cx(num_qubits - 1, num_qubits - 2)
228         feature_map.cx(0, num_qubits - 1)
229         for i in range(1, num_qubits - 1):
230             feature_map.cx(i, i-1)
231     return feature_map
232 
233 def variational_circuit(num_qubits=3):
234 
235     var_circuit = TwoLocal(num_qubits, ['ry', 'rz'], ['cx'], entanglement='linear', reps=4,
    insert_barriers=True)
236 
237 
238     # return the variational circuit
239     return var_circuit
240 
241 # draw circuit templates
242 features_maps = [feature_map_expr1, feature_map_expr2, feature_map_expr3, feature_map_expr4,
    feature_map_expr5, feature_map_expr6]
243 for fmap in features_maps:
244     fmp = fmap(num_qubits=3,reps=1)
245     fmp.draw(output='mpl',filename=str(fmap)+'.jpg')
246 
247 
248 
249 def wrap(data, k=5, feature_map=None, opt_func=None, var_circuit=None, opt_params={},
    maxiter=100, reps=2, num_qubits=3):
250     mds = DataSet(data, k=k, A=3, B=6)
251     log = "nqubits({}), reps {}, feature_map {}, opt_func {}:".format(
252         num_qubits,
253         reps,
254         feature_map.__name__,
255         opt_func.__name__
256     )
257     print("="*80+"\n"+log)
258     with open('result.log', 'a+') as fp:
259         result_log = '#'+log+'\n'
260         fp.writelines(result_log)
261         for d in mds.data_gen():
262             print(log+" fold {}...".format(d['k']))
263             run = Main(reps=reps, num_qubits=num_qubits, fm_func=feature_map,
    vc_func=var_circuit)
264             run.opt = opt_func(maxiter=maxiter, **opt_params)
265             acc = run.train(d['tr_input'], d['te_input'])
```

```
266                 fp.writelines("{}, {}\n".format(d['k'], acc))
267             run.drawfmap(log.replace('
    ','').replace(",","").replace(":","").replace("(","").replace(")",""))
268             run.drawvmap(log.replace('
    ','').replace(",","").replace(":","").replace("(","").replace(")",""))
269
270
271 newdata = data1.copy()
272 params = {"k":5, "maxiter":80, "num_qubits":newdata.shape[1]-1}
273 params["var_circuit"] = variational_circuit
274 params["opt_func"] = COBYLA
275 params["opt_params"] = {"disp":True, "tol":1e-6}
276 for fmap in features_maps:
277     for rep in [1,2,3,4]:
278         params["feature_map"] = fmap
279         params["reps"] = rep
280         wrap(newdata, **params)
```

VQC.py