Introduction to quantum computing: machine learning

Erik Spence

SciNet HPC Consortium

6 June 2023

Today's code and slides

You can get the slides and code for today's class at the SciNet Education web page.

`https://scinet.courses/1290`

Click on the link for the class, and look under "Lectures", click on "Machine learning".

This lecture

Suppose we're interested in applying machine learning algorithms to a quantum computer. How would we implement that? This class will examine how this is approached.

- Encoding data into a circuit.
- How to implement amplitude encoding.
- Variational classifier example.

This is just a brief introduction to a very active research area.

Quantum machine learning

Many quantum implementations of classical machine learning algorithms have been implemented:

- k-means clustering,
- Support vector machines,
- Neural networks,
- Principle component analysis,
- Eigenvalue solvers,
- Systems of linear equations.

And others. More are being developed all the time.

Encoding information

Grover's algorithm had the solution encoded directly into it. In contrast, machine learning models are usually 'trained' (at least supervised-learning models), which means that they will have parameters that need to be tuned to the data set.

But how do we get the data into the model? What approaches are there for encoding data so that the quantum machine-learning model can use it?

- basis encoding,
- amplitude encoding,
- angle encoding,
- dynamic encoding.

We'll discuss these in turn.

Encoding information: basis encoding

Basis encoding associates the n-qubit basis state with an n-bit number.

- This is exactly what classical computers do: $|\mathbf{5}\rangle = |\mathbf{0101}\rangle$.
- This is direct and simple, since each bit gets directly replaced by a qubit.
- The amplitudes of the basis states are not used as part of the algorithm.
- The goal of the algorithm will be to maximize the probability of measuring the correct answer, as represented by the final state.
- This approach obviously relies on some convention for what each bit represents, especially if floating-point numbers are being represented.

The resolution of this approach is obviously limited by the number of bits in the circuit.

Encoding information: amplitude encoding

Amplitude encoding associates the floating point number that is being represented with amplitudes of the resulting quantum states.

$$\mathbf{x} = [x_1 \ \ldots \ x_{2^n}]^T \implies |\psi_{\mathbf{x}}\rangle = \sum_{j=1}^{2^n} x_j \, |j\rangle$$

- Thus, continuous numbers can be encoded directly into the amplitudes of the qubits.
- Matrices can be similarly encoded.
- This approach, while appealing due to the information density, does have drawbacks.
- There are limits on what operations can be applied to the circuit. In particular nonlinear mappings of the amplitudes are often not compatible with unitary operators.
- Only normalized data can be processed. A degree of freedom must be sacrificed.

This is the approach which we will use in today's example.

Encoding information: angle encoding

Angle encoding associates the floating point number that is being represented with the angles of the quantum states.

$$\mathbf{x} = [x_1 \ \ldots \ x_{2^n}]^T \implies |\psi_{\mathbf{x}}\rangle = \bigotimes_{j=1}^{2^n} \left(\cos\left(x_j\right)|0\rangle + \sin\left(x_j\right)|1\rangle\right)$$

- Again, continuous numbers can be encoded directly into the amplitudes of the qubits.
- But only a single number can be encoded at a time, rather than the whole data set.

This approach only requires n qubits or less to encode.

Encoding information: dynamic encoding

Dynamic encoding refers to encoding data information into the operators, $\boldsymbol{A}$, themselves.

- If the operator $\boldsymbol{A}$ is unitary, then we're good to go.
- If $\boldsymbol{A}$ is not unitary then one can sometimes use the encoding

$$\tilde{\boldsymbol{A}} = \begin{bmatrix} \mathbf{0} & \boldsymbol{A} \\ \boldsymbol{A}^\dagger & \mathbf{0} \end{bmatrix}$$

  and then only use part of the output.
- With this approach, the eigenvalues of $\boldsymbol{A}$ can be processed within the circuit.

We won't revisit this approach today.

Implementing amplitude encoding

The first step to building a quantum machine learning model is encoding the data into the model. In today's class we will use amplitude encoding. How does that work?

$$\mathbf{x} = [x_1 \ \ldots \ x_{2^n}]^T \implies |\psi_\mathbf{x}\rangle = \sum_{j=1}^{2^n} x_j \, |j\rangle$$

Of course, the data set must be normalized such that $|x|^2 = 1$. Thus the goal is to create

$$|\psi\rangle = \sum_i x_i \, |i\rangle$$

under that condition.

Implementing amplitude encoding, continued

So we're after

$$|\psi\rangle = \sum_i x_i |i\rangle$$

with $\mathbf{x} = [x_1 \ \ldots \ x_{2^n}]^T$. How do can we build a circuit that will prepare such a state?

More generally, this problem can be posed as: how do we create a circuit that will map an arbitrary input state $|a\rangle$ to another arbitrary state $|b\rangle$?

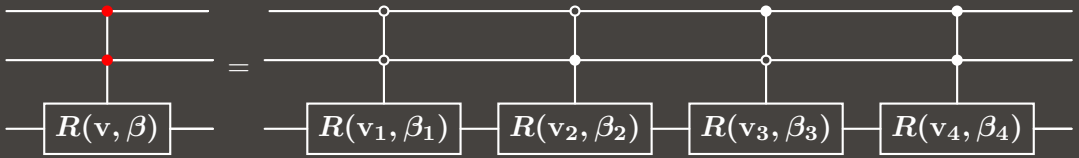
We can use such an algorithm to prepare an arbitrary input state.

Multi-controlled rotation gates

Before we introduce the algorithm we must first introduce the 'multi-controlled rotation' gate.

- It controls a rotation on qubit q_n by all possible states of the other qubits, $q_1, ..., q_{n-1}$.
- This means we will do a different rotation for every possible superposition of states.
- We will need a full set of rotations around vectors $\mathbf{v}_i$ by angles β_i.

Below is the case of $n = 3$.

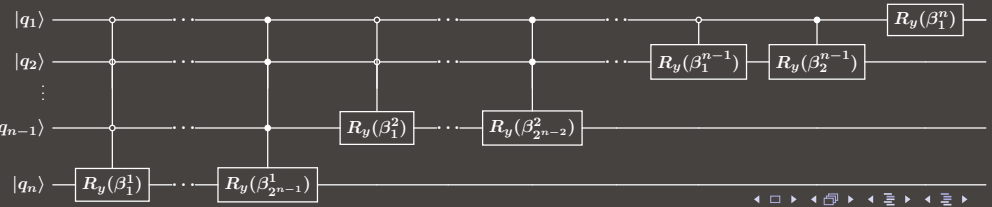


Implementing amplitude encoding, continued more

$$U\,|a\rangle = |b\rangle$$

Our algorithm will construct $|b\rangle$ from $|a\rangle$, where $|b\rangle = |0..0\rangle$ and $|a\rangle$ is an arbitrary initial state, by performing the following steps:

- We begin by applying an R_y multiple control rotation to qubit n, controlling on qubits $1 \to (n-1)$.
- We follow this by applying an R_y multiple control rotation to qubit $n-1$, controlling qubits $1 \to (n-2)$.
- We continue this pattern up to qubit 1.



Implementing amplitude encoding, continued

That circuit is weird. See the references on the last slide for details on where that came from.

But what about the values of β_j^s? The derivation of the previous circuit also allowed those values to be calculated:

$$\beta_j^n = 2\sin^{-1}\left(\frac{\sqrt{\sum_{\ell=1}^{2^{n-1}}\left|x_{(2j-1)2^{n-1}+\ell}\right|^2}}{\sqrt{\sum_{\ell=1}^{2^n}\left|x_{(j-1)2^n+\ell}\right|^2}}\right)$$

And there you have it. We now have all the information we need to prepare any given state.

Amplitude encoding, an example

Suppose we're after the state

$$|\psi\rangle = \sqrt{0.2}\,|000\rangle + \sqrt{0.45}\,|010\rangle + \sqrt{0.05}\,|011\rangle + \sqrt{0.2}\,|110\rangle + \sqrt{0.1}\,|111\rangle$$

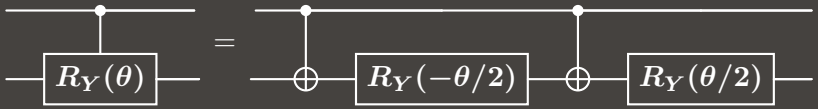
With $x_1 = \sqrt{0.2}$, $x_3 = \sqrt{0.45}$, $x_4 = \sqrt{0.05}$, $x_7 = \sqrt{0.2}$, $x_8 = \sqrt{0.1}$.

This gives us the corresponding rotations angles: $\beta_1^1 = 0$, $\beta_2^1 = 0.644$, $\beta_3^1 = 0$, $\beta_4^1 = 1.231$, $\beta_1^2 = 2.014$, $\beta_2^2 = 3.142$, $\beta_1^3 = 1.159$.

With these values the corresponding operators can be built. But the algorithm is going from $|a\rangle$ to $|b\rangle = |0..0\rangle$. To go the other way around we need merely invert these operations.

Controlled R_Y

Yesterday's optional hands-on asked you to implement a controlled R_Y gate using CNOT gates and single-qubit R_y gates. The solution is below.



We will see this presently, as we see an implementation of a 2-qubit amplitude-encoding circuit.

Amplitude encoding data, by hand

```
# encoding.py

import pennylane as qml

dev = qml.device('default.qubit', wires = 2)

def get_angles(x):
  # Calculate the beta values.
  beta1_1 = 2 * np.arcsin(np.sqrt(x[1]**2) /
            np.sqrt(x[0]**2 + x[1]**2 + 1e-12))
  beta1_2 = 2 * np.arcsin(np.sqrt(x[3]**2) /
            np.sqrt(x[2]**2 + x[3]**2 + 1e-12))
  beta2_1 = 2 * np.arcsin(np.sqrt(x[2]**2 + x[3]**2) /
            np.sqrt(x[0]**2 + x[1]**2 + x[2]**2 + x[3]**2))
                    x[2]**2 + x[3]**2))

  return beta1_1, beta1_2, beta2_1
```

```
@qml.qnode(dev)
def statepreparation(a):
  qml.RY(a[2], wires = 0)

  qml.CNOT(wires = [0, 1])
  qml.RY(-a[1]/2, wires = 1)
  qml.CNOT(wires = [0, 1])
  qml.RY(a[1]/2, wires = 1)

  qml.PauliX(wires = 0)
  qml.CNOT(wires = [0, 1])
  qml.RY(-a[0]/2, wires = 1)
  qml.CNOT(wires = [0, 1])
  qml.RY(a[0]/2, wires = 1)
  qml.PauliX(wires = 0)

  return qml.state()
```

Amplitude encoding data, with Pennylane

But as you might expect, encoding data in this way is tedious. Is there not a better way?

Of course there is. The ability to prepare you data into an arbitrary amplitude encoding is built into PennyLane.

Either the input needs to be normalized, or the normalize = True flag needs to be set.

```
In [1]: import encoding as en
In [2]:
In [2]: angles = en.get_angles([1, 2, 3, 4])
In [3]:
In [3]: en.statepreparation(angles)
Out[3]: tensor([0.18257419+0.j, 0.36514837+0.j,
         0.54772256+0.j, 0.73029674+0.j], requires_grad=True)
In [4]:
In [4]: import pennylane as qml
In [5]:
In [5]: qml.AmplitudeEmbedding([1, 2, 3, 4], [0, 1],
...:                           normalize = True)
Out[5]: AmplitudeEmbedding(array([0.18257419+0.j,
             0.36514837+0.j, 0.54772256+0.j,
             0.73029674+0.j]), wires=[0, 1])
In [6]:
```

Variational classifier example

Now that we've determined a means of encoding our data into our quantum circuit, let's do a supervised-learning example, meaning an example where we have input values $\mathbf{x}$ ('features'), and 'target' values $\boldsymbol{y}$.

We will build a variational classifier.

- A variational classifier is a classification model that is based on free parameters.
- A cost function is built that measures how badly the model is predicting the target values.
- The cost function is minimized, outside the quantum circuit, by varying the free parameters, thus optimizing the model.

If this sounds like the approach used to train a neural network, it's because they are closely related.

Variational classifier example, continued

We will build a classifier that will classify the 'occupancy' data set.

- The data describes occupancy characteristics of houses. The target (or 'label') is whether or not the given house is occupied.
- The data set has been reduced to 4 features, and the binary label.
- As such, we can encode the 4 features into 2 qubits.
- The label has been cast as $+1$/-1. We will use the Pauli Z operator as our output observable.

The original source for the data set is here:
`https://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+`

Dealing with the data

Let's first deal with preparing the data.

We normalize all the columns so that one column does not completely dominate the others when they are fed into the circuit.

We do the train-test split in the usual way.

```
import sklearn.preprocessing as skpp
import sklearn.preprocessing as skms

def get_data():

  data = np.loadtxt('occupancy_small.csv',
                    delimiter = ',')
  num_feat = data.shape[1] - 1
  features = data[:, 0:num_feat]

  features = skpp.normalize(features, axis = 0)

  y = data[:, -1]

  return skms.train_test_split(features, y,
                               test_size = 0.2)
```

Building our model

```
import pennylane as qml
from pennylane import numpy as np

dev = qml.device("default.qubit", wires = 2)

@qml.qnode(dev, interface = "autograd")
def circuit(weights, x, wires):

  # Prepare the state of the data.
  qml.AmplitudeEmbedding(x, wires,
                         normalize = True)

  # Add the layers.
  for W in weights: layer(W)

  # Return the expectation value of Pauli Z.
  return qml.expval(qml.PauliZ(0))
```

```
def layer(W):
  # Rotate the state by three angles.
  qml.Rot(W[0, 0], W[0, 1], W[0, 2], wires = 0)
  qml.Rot(W[1, 0], W[1, 1], W[1, 2], wires = 1)

  # Entangle the qubits.
  qml.CNOT(wires = [0, 1])
```

The model circuit consists of several steps:

- Amplitude-encode the data into the circuit.
- Add layers. These consist of to-be-determined rotations (θ, ϕ, ω).
- The expectation value of the Pauli Z operator is given, indicating either a +1 or -1 value.

Building our model, continued

```python
def accuracy(y_true, y_pred):

  # Initialize the loss.
  loss = 0

  # Loop over the data.
  for t, p in zip(y_true, y_pred):

    # If this difference is small,
    # increment.
    if abs(t - p) < 1e-5:
      loss = loss + 1

    # Normalize.
    loss = loss / len(y_true)

  return loss
```

```python
def cost(weights, x, y, wires):

  # Calculate the predictions for all the data.
  y_pred = [circuit(weights, this_x, wires)
            for this_x in x]

  # Initialize the loss.
  loss = 0

  # Loop over the data, add the difference squared.
  for t, p in zip(y_true, y_pred):
    loss = loss + (t - p)**2

  # Return the normalized value.
  return loss / len(labels)
```

Building our model, continued more

```
import pennylane.numpy.random as npr
import pennylane.optimize as po

# Get the data.
x_train, x_test, y_train, y_test = get_data()

# Initialize some parameters.
num_qubits = 2
num_layers = 4
wires = range(num_qubits)
batch_size = 5
num_train = len(y_train)

# Initialize the weights to be optimized.
weights = 0.01 * npr.randn(num_layers,
          num_qubits, 3, requires_grad = True)
```

```
# Initialize the optimizer.
opt = po.NesterovMomentumOptimizer(0.01)
for it in range(100):

  # Select a batch of data.
  batch_index = npr.randint(0, num_train,
                            (batch_size,))
  x_batch = x_train[batch_index]
  y_batch = y_train[batch_index]

  # Update the weights.
  step = opt.step(cost, weights, x_batch,
                  y_batch, wires)
  weights = step[0]

  # Print out accuracy every so often.
```

Running the example

```
ejspence@mycomp ~>
ejspence@mycomp ~> python occupancy_vc.py Getting data
Iter:  0 | Cost:  2.3622870 | Acc train:  0.0400000 | Acc test:  0.0550000
Iter:  10 | Cost:  1.4128026 | Acc train:  0.7570000 | Acc test:  0.7900000
Iter:  20 | Cost:  0.2122022 | Acc train:  0.9680000 | Acc test:  0.9550000
Iter:  30 | Cost:  0.2100777 | Acc train:  0.9600000 | Acc test:  0.9550000
Iter:  40 | Cost:  0.1685274 | Acc train:  0.9690000 | Acc test:  0.9600000
Iter:  50 | Cost:  0.1643545 | Acc train:  0.9670000 | Acc test:  0.9550000
Iter:  60 | Cost:  0.1812916 | Acc train:  0.9550000 | Acc test:  0.9500000
Iter:  70 | Cost:  0.1617029 | Acc train:  0.9630000 | Acc test:  0.9550000
Iter:  80 | Cost:  0.1619873 | Acc train:  0.9670000 | Acc test:  0.9550000
Iter:  90 | Cost:  0.1634609 | Acc train:  0.9680000 | Acc test:  0.9600000
Iter:  100 | Cost:  0.1816448 | Acc train:  0.9750000 | Acc test:  0.9650000
ejspence@mycomp ~>
```

Notes about our model

This is a good start. We successfully classified the data.

- This approach will only work for binary classification. Having more than two categories will require a different approach.
- If the model seems ad hoc, that's because it it. But that's ok, as long as the model works.
- The optimization of the parameters is done outside of the model. There's nothing quantum about that.
- The model, as we built it, can only handle a single data point at a time.
- Consequently, evaluating the accuracy of the model is slow, since the model needs to be rebuilt for each new data point.

Nonetheless, this represents a good start.

Summary

A review of this lecture's material.

- Using a quantum computer to build a machine-learning model requires some means of passing the data into the model.
- Amplitude encoding is one such approach. It involves encoding the values of the data into the amplitudes of individual quantum states.
- Once so encoded, we can build a simple variational classifier into our circuit.
- The parameters of the circuit are trained using a gradient-based approach, which is built into PennyLane.
- The final model worked well, but was slow to train.

The point of this exercise was to introduce the machinery that is sometimes used to build such models.

Quantum machine learning: hands on

There are many aspects of the model that you can vary to improve the model's performance:

- Choose fewer layers, to make things run faster, and perhaps reduce overfitting.
- Use more data, to improve the accuracy of the model.
- Add another qubit, to encode two data points at once, rather than just one.

Play around with these options and see if you can improve your model's final result.

Linky goodness

Data encoding:

- https://arxiv.org/abs/quant-ph/0404089
- https://arxiv.org/abs/quant-ph/0407010
- https://pennylane.ai/blog/2022/08/how-to-embed-data-into-a-quantum-state

Quantum machine learning:

- https://link.springer.com/book/10.1007/978-3-319-96424-9
- https://arxiv.org/abs/1806.06871
- https://pennylane.ai/qml/demos/tutorial_variational_classifier.html