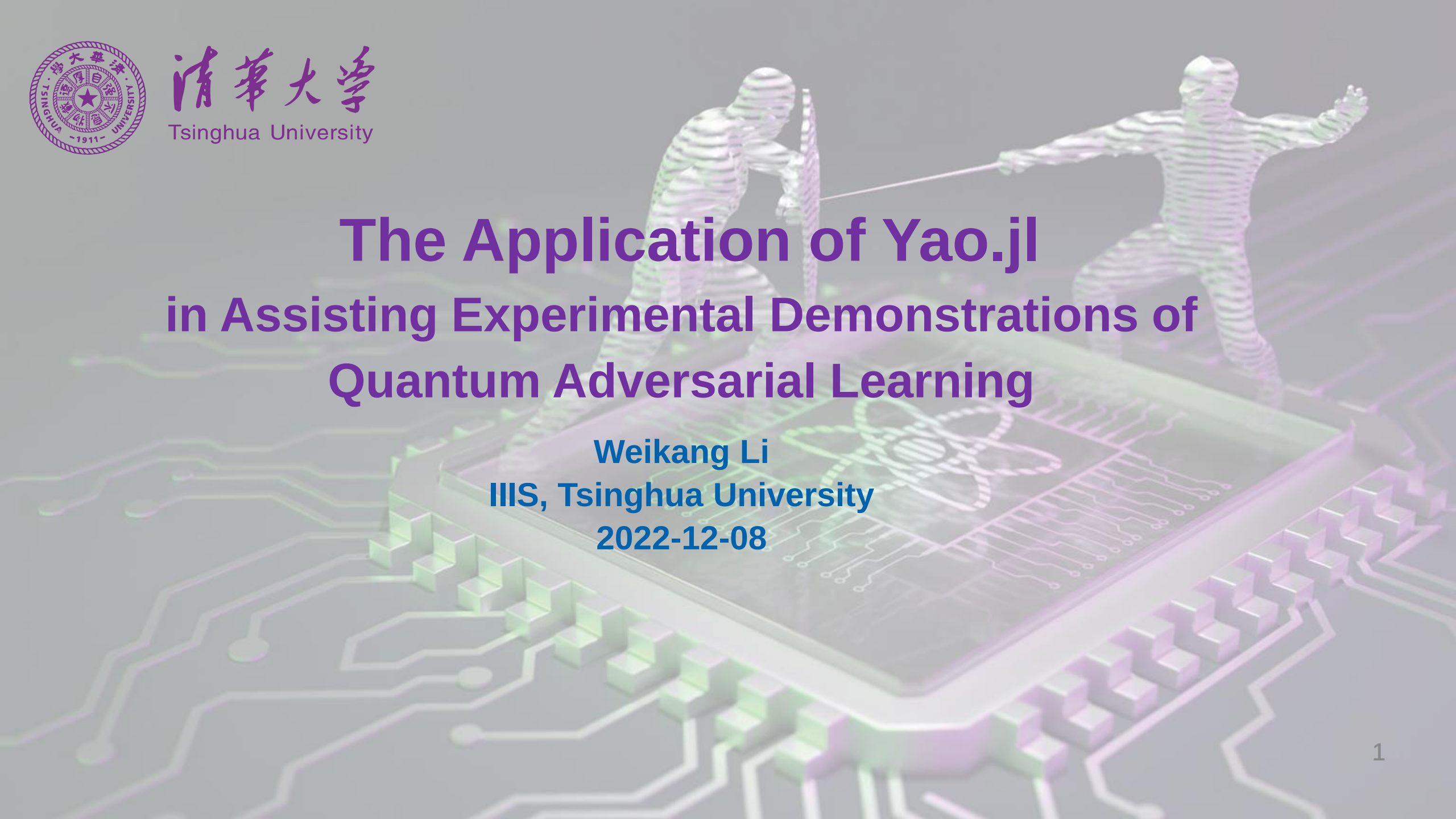


The Application of Yao.jl in Assisting Experimental Demonstrations of Quantum Adversarial Learning

Weikang Li
IIIS, Tsinghua University
2022-12-08

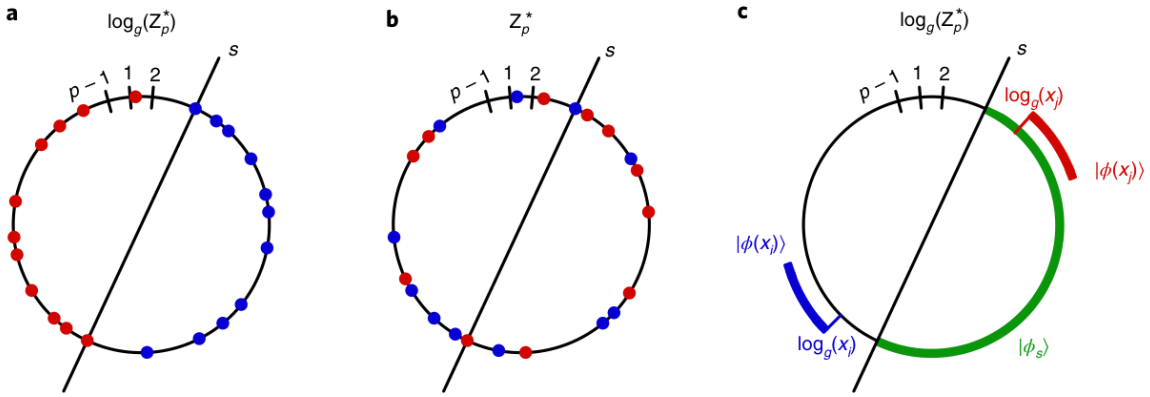
Quantum machine learning

Rigorous quantum speedup in classification tasks:

nature physics
ARTICLES
https://doi.org/10.1038/s41567-021-01287-z
Check for updates

A rigorous and robust quantum speed-up in supervised machine learning

Yunchao Liu[1,2], Srinivasan Arunachalam[2] and Kristan Temme[2]✉



Based on DLP, we define our concept class $\mathcal{C} = \{f_s\}_{s\in\mathbb{Z}_p^*}$ over the data space $\mathcal{X} = \mathbb{Z}_p^*$ as follows:

$$f_s(x) = \begin{cases} +1, & \text{if } \log_g x \in [s, s + \frac{p-3}{2}], \\ -1, & \text{else.} \end{cases} \tag{5}$$

Theorem 1. *Assuming the classical hardness of DLP, no efficient classical algorithm can achieve* $\frac{1}{2} + \frac{1}{\text{poly}(n)}$ *test accuracy for* $\mathcal{C}$.

Theorem 2. *The concept class* $\mathcal{C}$ *is efficiently learnable by SVM-QKE. More specifically, for any concept* $f \in \mathcal{C}$*, the SVM-QKE algorithm returns a classifier with test accuracy at least 0.99 in polynomial time, with probability at least 2/3 over the choice of random training samples and over noise in QKE estimation.*

Quantum machine learning

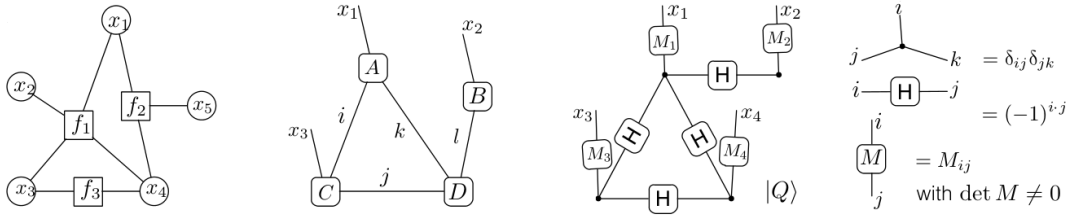
Rigorous quantum speedup in generative models:

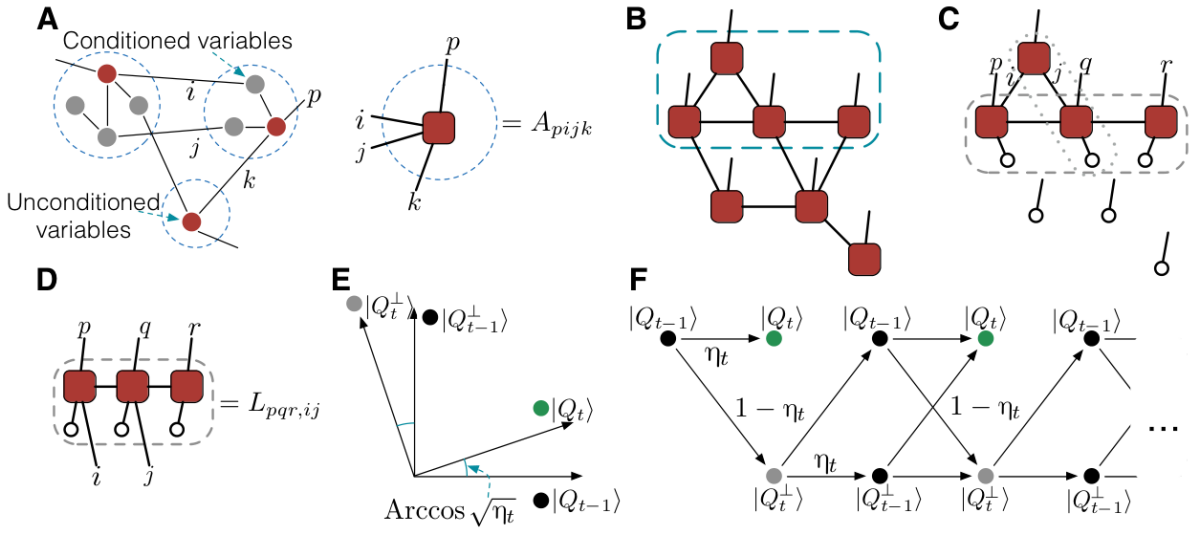
SCIENCE ADVANCES | RESEARCH ARTICLE

PHYSICS

A quantum machine learning algorithm based on generative models

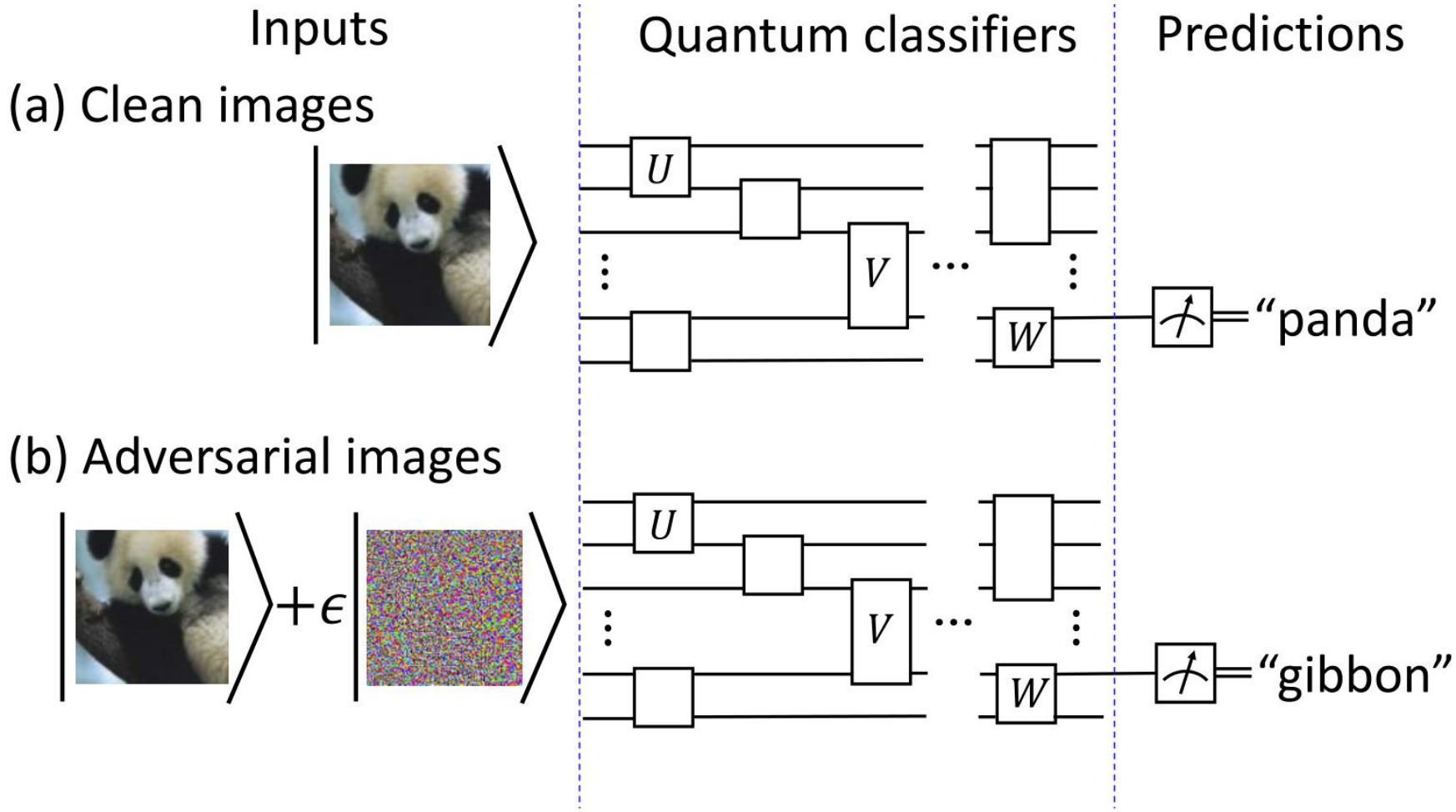
X. Gao[1]*, Z.-Y. Zhang[1,2], L.-M. Duan[1,2†]





In this paper, the authors defined an optimization task of a generative model with the Kullback-Leibler divergence as the cost function. They successfully mapped the parameter training task to the preparation of a tensor network state |Q(z)⟩. Then it was proved that, there exist instances for “computing the gradients” of the cost function up to an additive error where the quantum algorithm could provide a polynomial time solution, while any classical algorithm cannot solve it in polynomial time as long as universal quantum computers cannot be efficiently simulated by classical ones.

Quantum adversarial learning



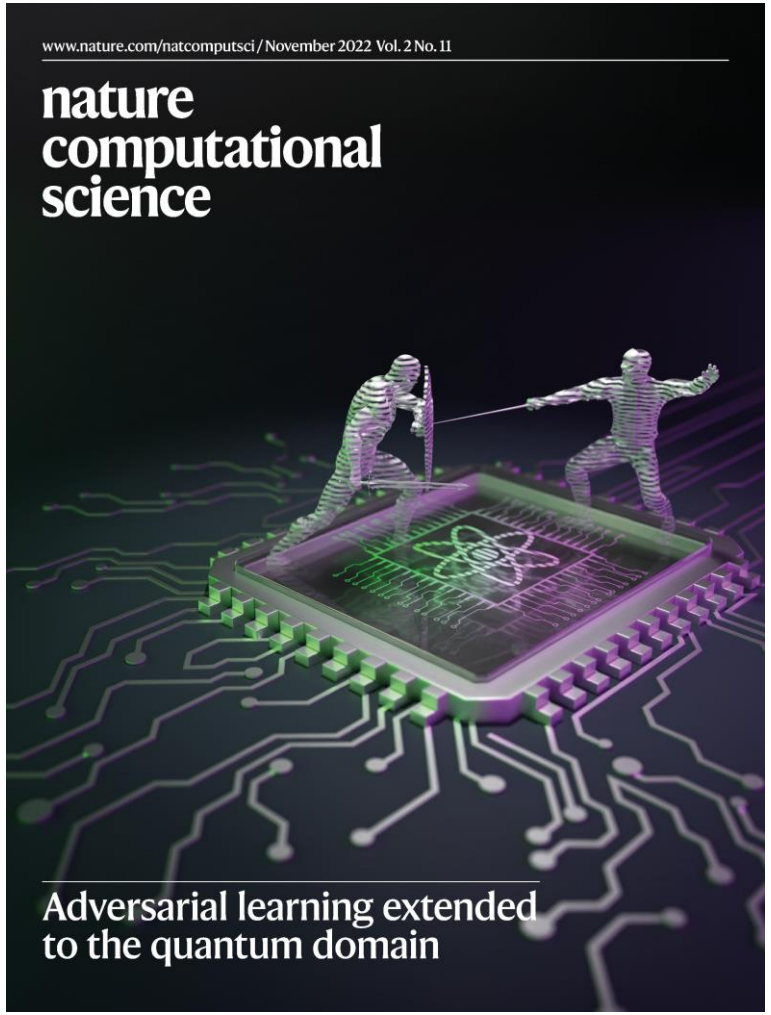
Lu, Duan, and Deng, Phys. Rev. Research 2 (3), 033212

Questions & Motivations

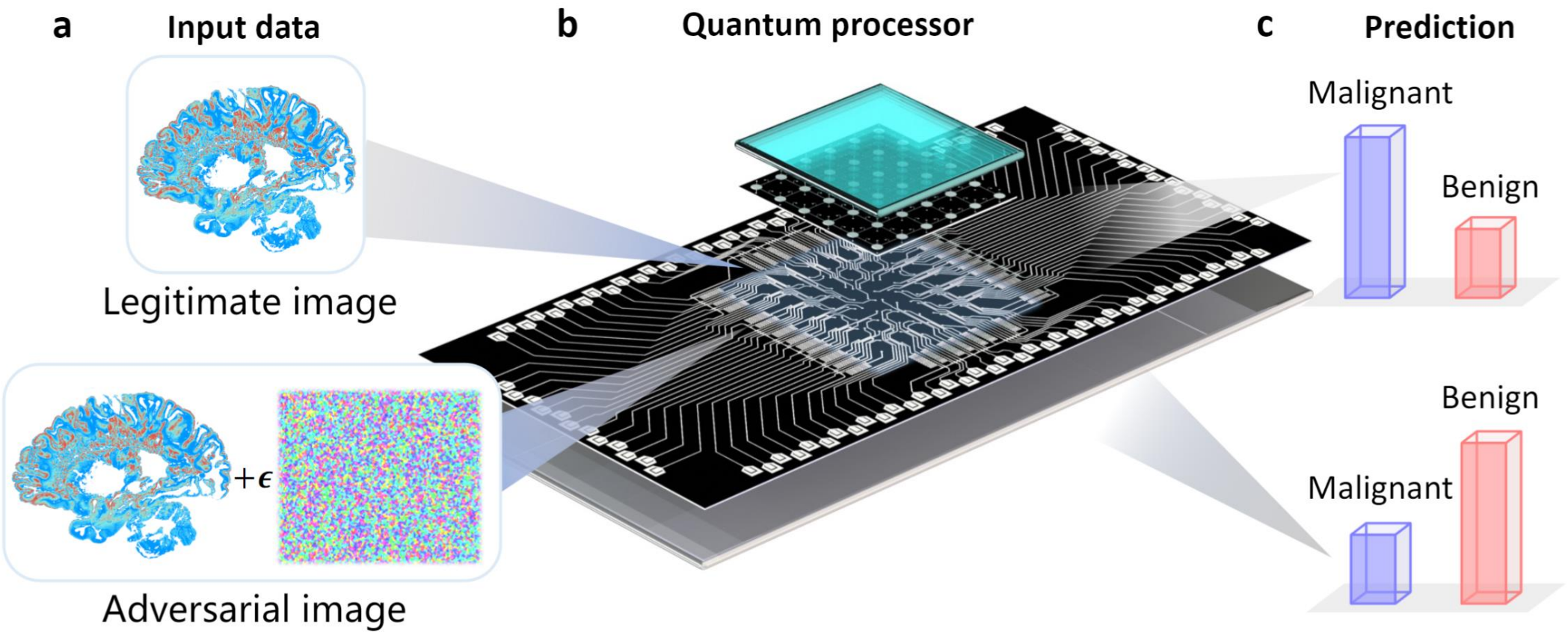
- Can we design an experimental quantum classification model such that, considering the unavoidable experimental noise, it can still handle the classification of high-dimensional and real-life datasets? (as a basis for the followed adversarial learning)

- Are quantum learning models on real quantum devices vulnerable to adversarial noises? (In classical machine learning, adding noise to images is a way to defend against adversarial attacks; Besides, according to Du et al, the quantum noise may protect quantum classifiers against adversaries)

- If so, how can we design a defense strategy?



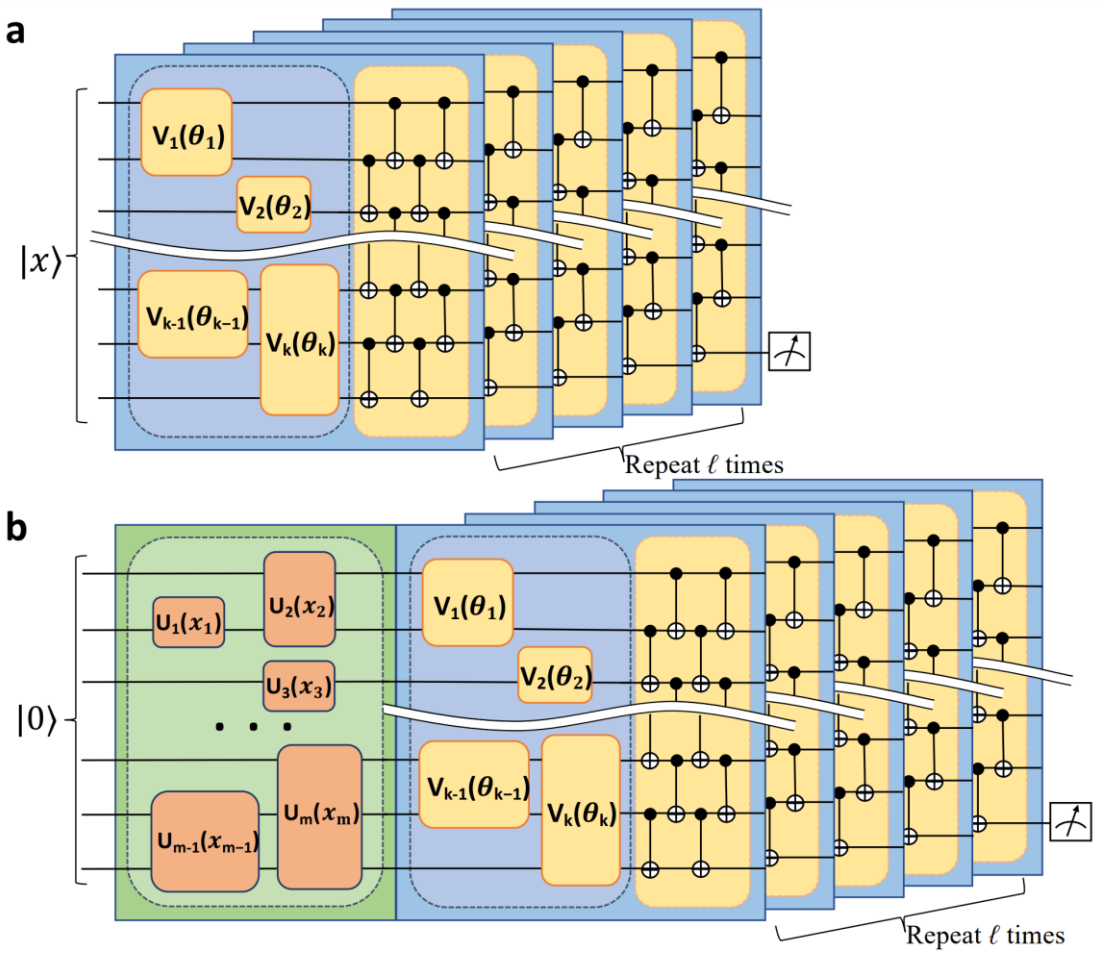
Superconducting platforms



Task 1:
Handling the classification of high-dimensional and real-life data

Model design & training

Data encoding:



For both methods shown in **a** and **b** (left), the data is encoded first, followed by a variational quantum circuit.

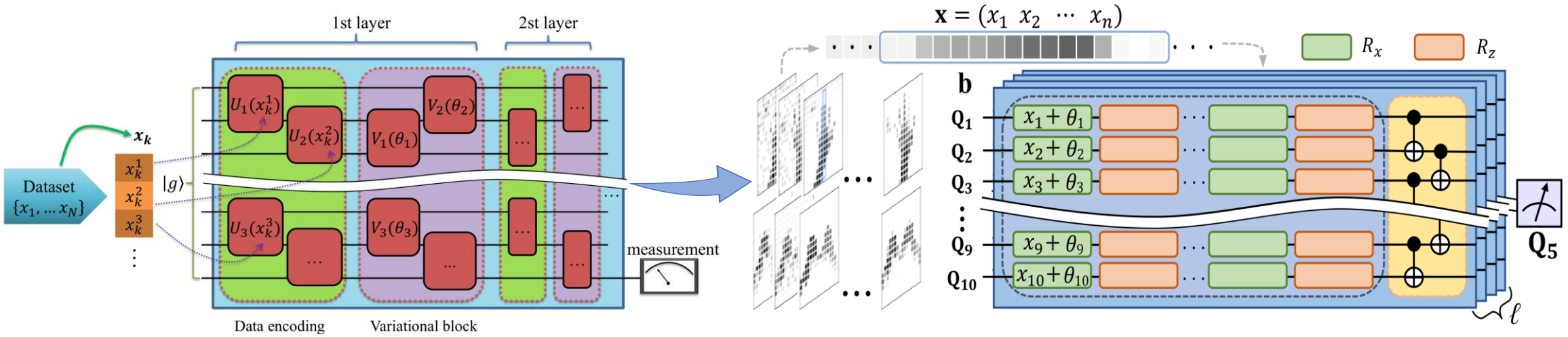
In this encoding-first strategy, once the data is encoded, the following learning process can be reduced to a support vector machine with kernel elements $\kappa(x, x') = |\langle\phi(x') \mid \phi(x)\rangle|^2$.

For a real-life dataset, it is **difficult** to design a general encoding-first strategy such that the kernel is "good" for classification.

Can we design a kernel method whose kernel can be variationally optimized as well?

Model design & training

Data encoding:



With this “interleaved” encoding structure, the corresponding kernel can be expressed as

$$\mathcal{K}'_{ij} = \left|\left\langle 0 \left| V'^{\dagger}_{\mathbf{x}_i,\boldsymbol{\theta}} V'_{\mathbf{x}_j,\boldsymbol{\theta}} \right| 0 \right\rangle\right|^2$$

, which can be adaptively optimized.

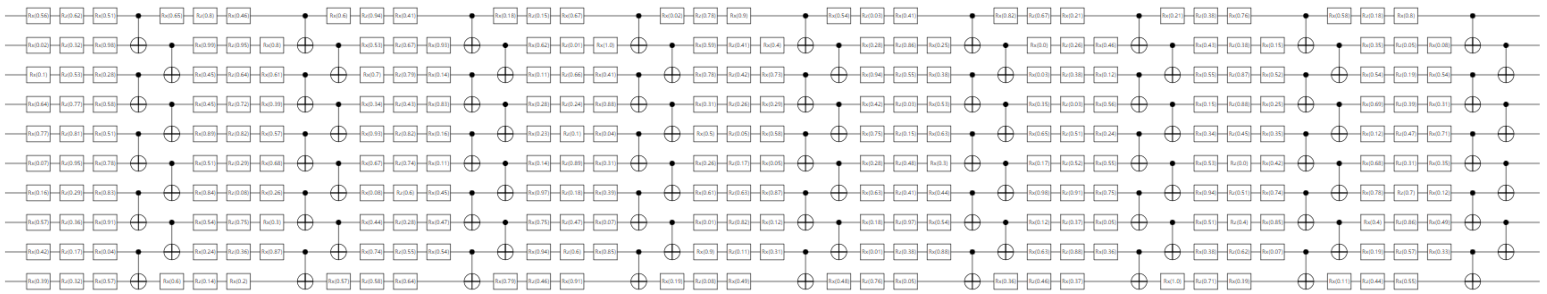
Model design & training

basic structure for block-encoding based QNNs

In [2]:

```
# import the FashionMNIST data
vars = matread("../dataset/FashionMNIST_1_2_wk.mat")
x_train = vars["x_train"]
y_train = vars["y_train"]
x_test = vars["x_test"]
y_test = vars["y_test"]
num_qubit = 10

# set the size of the training set and the test set
num_train = 500
num_test = 100
# set the scaling factor c = 2
c = 2
x_train = real(x_train[:,1:num_train])*c
y_train = y_train[1:num_train,:]
x_test = real(x_test[:,1:num_test])*c
y_test = y_test[1:num_test,:];

# define the QNN circuit, some functions have been defined before
depth = 9
circuit = chain(chain(num_qubit, params_layer(num_qubit),
        ent_cx(num_qubit)) for _ in 1:depth)
# assign random initial parameters to the circuit
dispatch!(circuit, :random)
# record the initial parameters
ini_params = parameters(circuit);
YaoPlots.plot(circuit)
```

Out[2]:

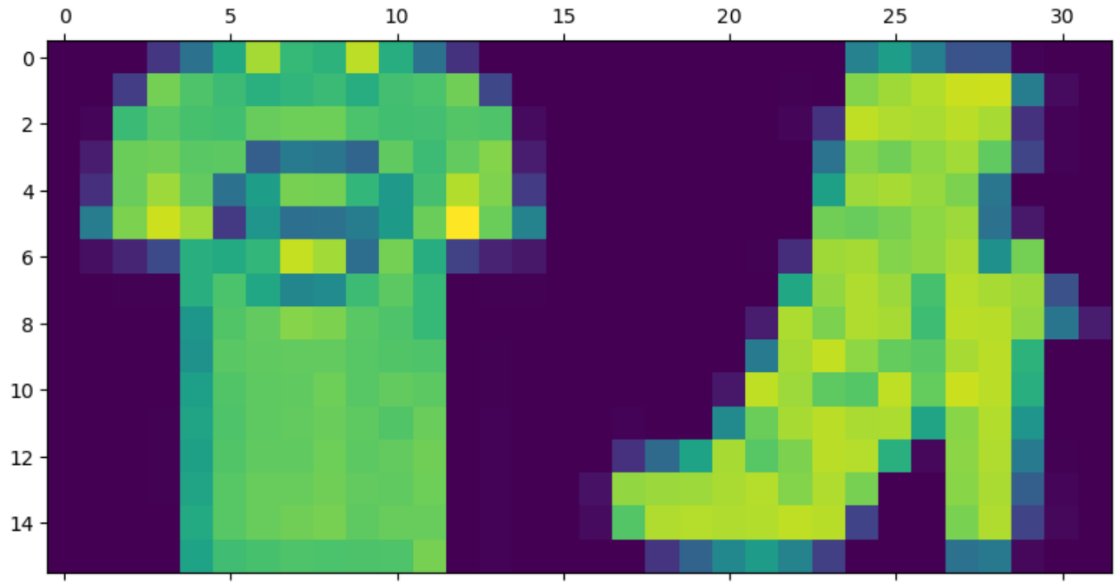


Model design & training

data visualization

In [3]:

```
i = 13
a = real(vars["x_train"][1:256,i])
c1 = reshape(a, (16,16))
i = 6
a = real(vars["x_train"][1:256,i])
c2 = reshape(a, (16,16))
matshow(hcat(c1,c2)) # T-shirt and ankle boot
```



data encoding

In [4]:

```
# we illustrate the idea of block-encoding based QNNs through a simple example
# the FashionMNIST dataset has been resized to be 256-dimensional
# we expand them to 270-dimensional by adding zeros at the end of the vectors
dim = 270
x_train_ = zeros(Float64, (dim,num_train))
x_train_[1:256,:] = x_train
x_train = x_train_
x_test_ = zeros(Float64, (dim,num_test))
x_test_[1:256,:] = x_test
x_test = x_test_

# the input data and the variational parameters are interleaved
# this strategy has been applied to [Ren et al, Experimental quantum adversarial
# learning with programmable superconducting qubits, arXiv:2204.01738]
# later we will numerically test the expressive power of this encoding strategy
train_cir = [chain(chain(num_qubit, params_layer(num_qubit), ent_cx(num_qubit))
        for _ in 1:depth) for _ in 1:num_train]
test_cir = [chain(chain(num_qubit, params_layer(num_qubit), ent_cx(num_qubit))
        for _ in 1:depth) for _ in 1:num_test];
for i in 1:num_train
    dispatch!(train_cir[i], x_train[:,i]+ini_params)
end
for i in 1:num_test
    dispatch!(test_cir[i], x_test[:,i]+ini_params)
end
```

Model design & training

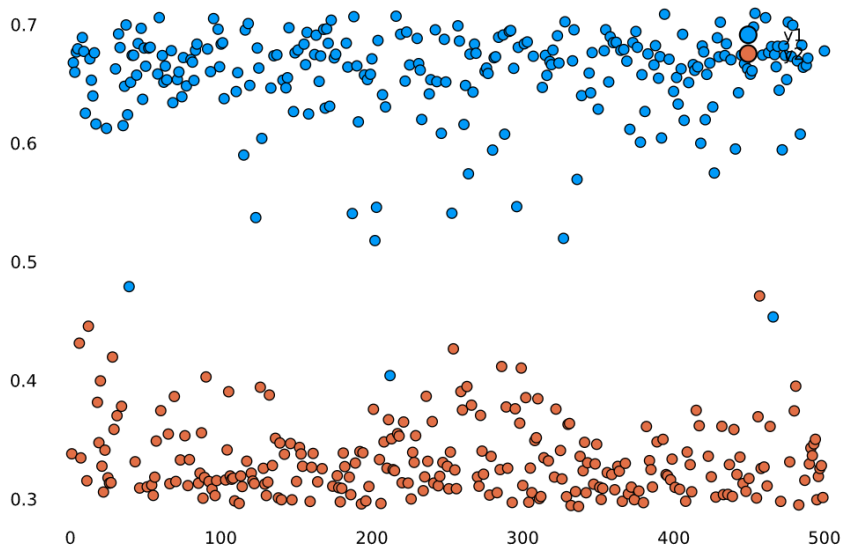
In [6]:

```
for k in 1:niters
    # calculate the accuracy & loss for the training & test set
    train_acc, train_loss = acc_loss_evaluation(num_qubit,train_cir,y_train,num_train, pos_)
    test_acc, test_loss = acc_loss_evaluation(num_qubit,test_cir,y_test,num_test, pos_)
    push!(loss_train_history, train_loss)
    push!(loss_test_history, test_loss)
    push!(acc_train_history, train_acc)
    push!(acc_test_history, test_acc)
    if k % 10 == 0
        @printf("\nStep=%d, loss=%.3f, acc=%.3f, test_loss=%.3f,test_acc=%.3f\n",k,train_loss,train_acc,test_loss,test_
    end

    # at each training epoch, randomly choose a batch of samples from the training set
    batch_index = randperm(size(x_train)[2])[1:batch_size]
    batch_cir = train_cir[batch_index]
    y_batch = y_train[batch_index,:]

    # this part is to add noise to test the robustness of the classifier
    # noise_batch_ = ε*rand(d, (batch_size,size(parameters(circuit))[1]))
    # for (i,j) in zip(batch_cir,1:batch_size)
    #     dispatch!(i,parameters(i)+noise_batch_[j,:])
    # end

    # for all samples in the batch, repeatly measure their qubits at position pos_
    # on the computational basis
    q_ = zeros(batch_size,2);
    for i=1:batch_size
        q_[i,:] = density_matrix(zero_state(num_qubit) |> batch_cir[i], (pos_)) |> probs
    end

    # calculate the gradients w.r.t. the cross-entropy loss function
    Arr = Array{Float64}(zeros(batch_size,nparameters(batch_cir[1])))
    for i in 1:batch_size
        Arr[i,:] = expect'(op0, zero_state(num_qubit)=>batch_cir[i])[2]
    end
    C = [Arr, -Arr]
    grads = collect(mean([-sum([y_batch[i,j]*((1 ./ q_)[i,j])*batch(C)[i,:,j] for j in 1:2]) for i=1:batch_size]))

    # noise_grad = 0.1*ε*rand(d, size(parameters(circuit))) # added to the gradient when testing the robustness
    # update the parameters
    updates = Flux.Optimise.update!(optim, copy(ini_theta), grads);
    ini_theta = updates

    # if the noise is added above, cancel it at the end
    # recover the x
    #for (i,j) in zip(batch_cir,1:batch_size)
    #    dispatch!(i,parameters(i)-noise_batch_[j,:])
    #end

    # update the parameters
    for i in 1:num_train
        dispatch!(train_cir[i], x_train[:,i]+ini_theta)
    end
    for i in 1:num_test
        dispatch!(test_cir[i], x_test[:,i]+ini_theta)
    end
end
```

In [9]:

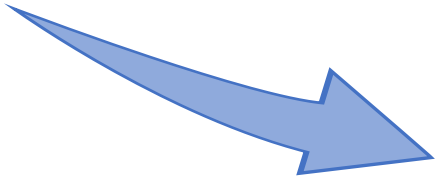
```
q_ = zeros(num_train,2);
for i=1:num_train
    q_[i,:] = density_matrix(zero_state(num_qubit) |> train_cir[i], (pos_)) |> probs
end
class1x = Int64[]
class2x = Int64[]
class1y = Float64[]
class2y = Float64[]
for i in 1:num_train
    if y_train[i,1] == 1.0
        push!(class1x,i)
        push!(class1y,q_[i,1])
    else
        push!(class2x,i)
        push!(class2y,q_[i,1])
    end
end
# predicted value (expectation value)
# lower loss leads to larger separation between the two classes of data points
Plots.plot(class1x, class1y, seriestype = :scatter)
Plots.plot!(class2x, class2y, seriestype = :scatter)
```

Out[9]:



Model design & training

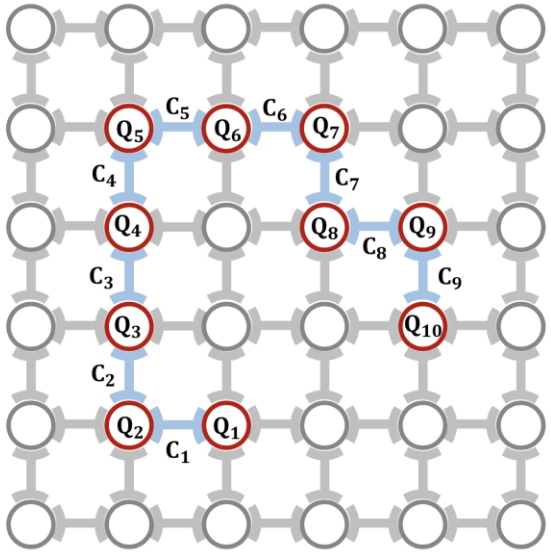
Alternative:
Parameter shift rule

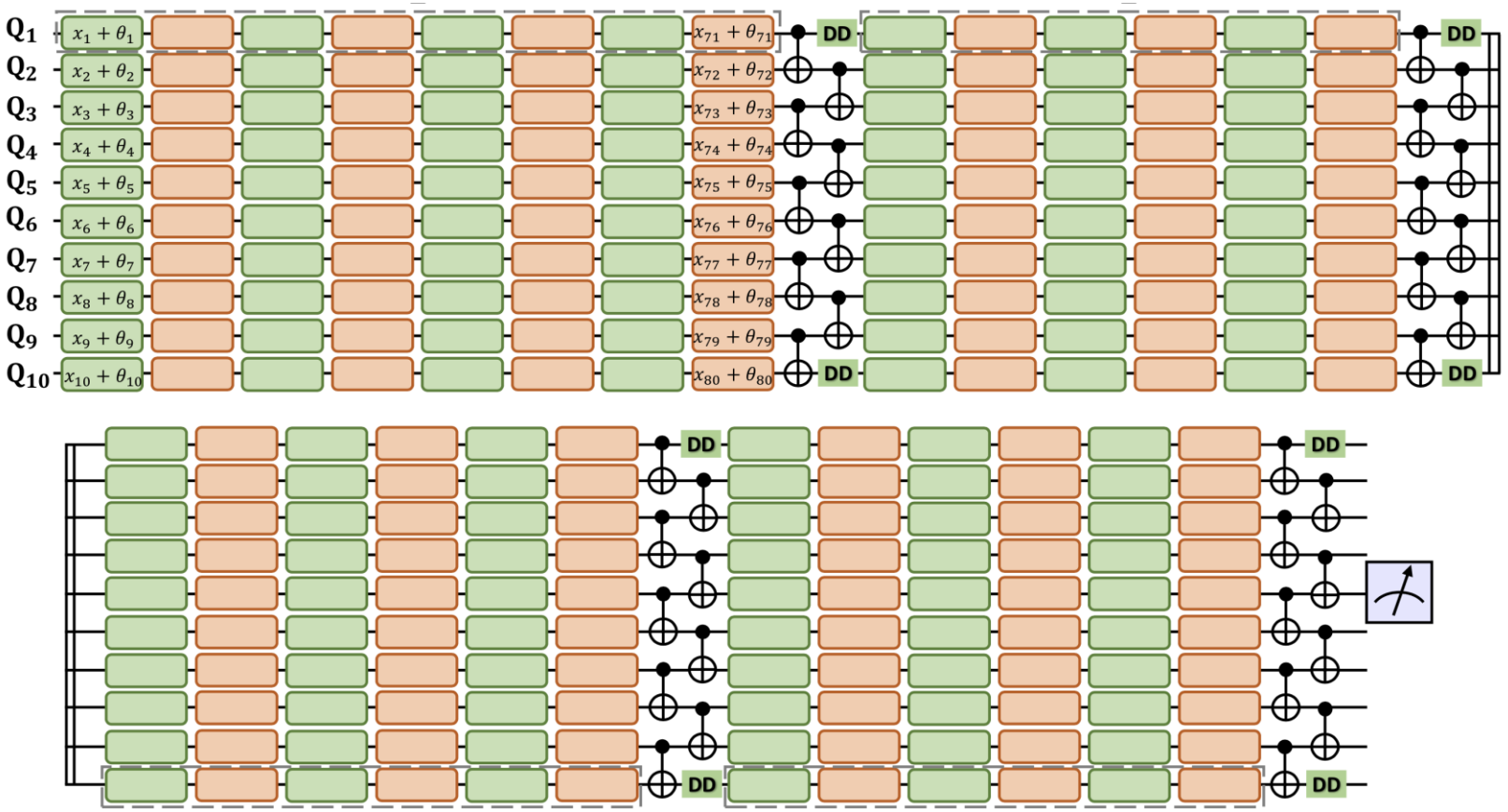


```
function opdiff(psifunc, diffblock::Diff)
    reg1, reg2 = _perturb(()-> psifunc(), diffblock, π/2)

    r0_1 = expect(op0, reg1) |> real;
    r1_1 = expect(op1, reg1) |> real;

    r0_2 = expect(op0, reg2) |> real;
    r1_2 = expect(op1, reg2) |> real;

    hcat((r0_2-r0_1)/2, (r1_2-r1_1)/2)
end

@inline function _perturb(func, gate::Diff{<:DiffBlock}, δ::Real)
    dispatch!(-, gate, (δ,))
    reg1 = func()
    dispatch!(+, gate, (2δ,))
    reg2 = func()
    dispatch!(-, gate, (δ,))
    reg1, reg2
end

@inline function _perturb(func, gate::Diff{<:Rotor}, δ::Real)
    dispatch!(-, gate, (δ,))
    reg1 = func()
    dispatch!(+, gate, (2δ,))
    reg2 = func()
    dispatch!(-, gate, (δ,))
    reg1, reg2
end
```

Model design & training

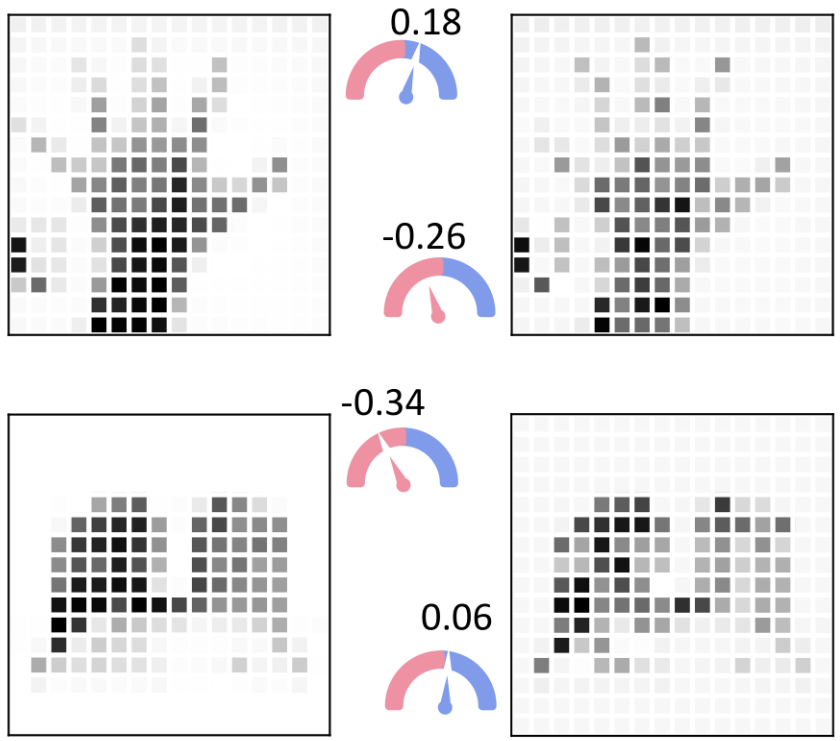
Experimental setup:

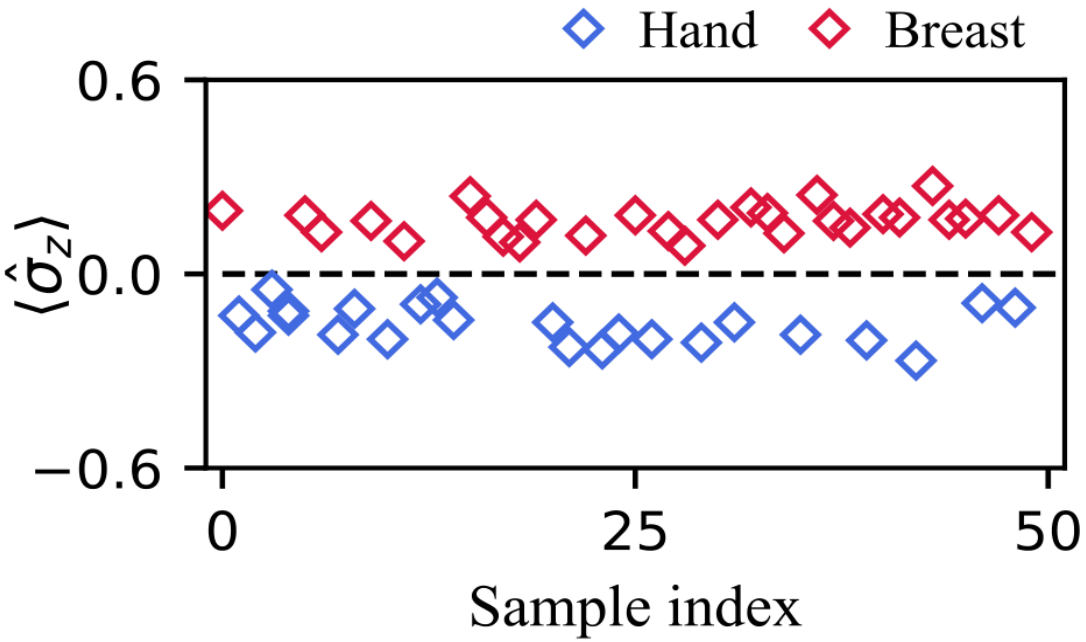




Model design & training

With the experimentally trained quantum classifier obtained before, we use the generated adversarial examples as inputs to see what results will be predicted:





Task 2: Handling the classification of many-body quantum states

Model design & training

The Aubry-André Hamiltonian:

$$H/\hbar = -\frac{g}{2}\sum_k \left(\hat{\sigma}_k^x \hat{\sigma}_{k+1}^x + \hat{\sigma}_k^y \hat{\sigma}_{k+1}^y\right) - \sum_k \frac{V_k}{2}\hat{\sigma}_k^z,$$

where g is the coupling strength, $\hat{\sigma}_k^l (l = x, y, z)$ is the Pauli operator for the k-th qubit, and $V_k = V\cos(2\pi\alpha k + \phi)$ is the incommensurate potential with V being the disorder magnitude, $\alpha = (\sqrt{5}-1)/2$ being an irrational number and ϕ being a random phase evenly distributed on $[0, 2\pi)$.

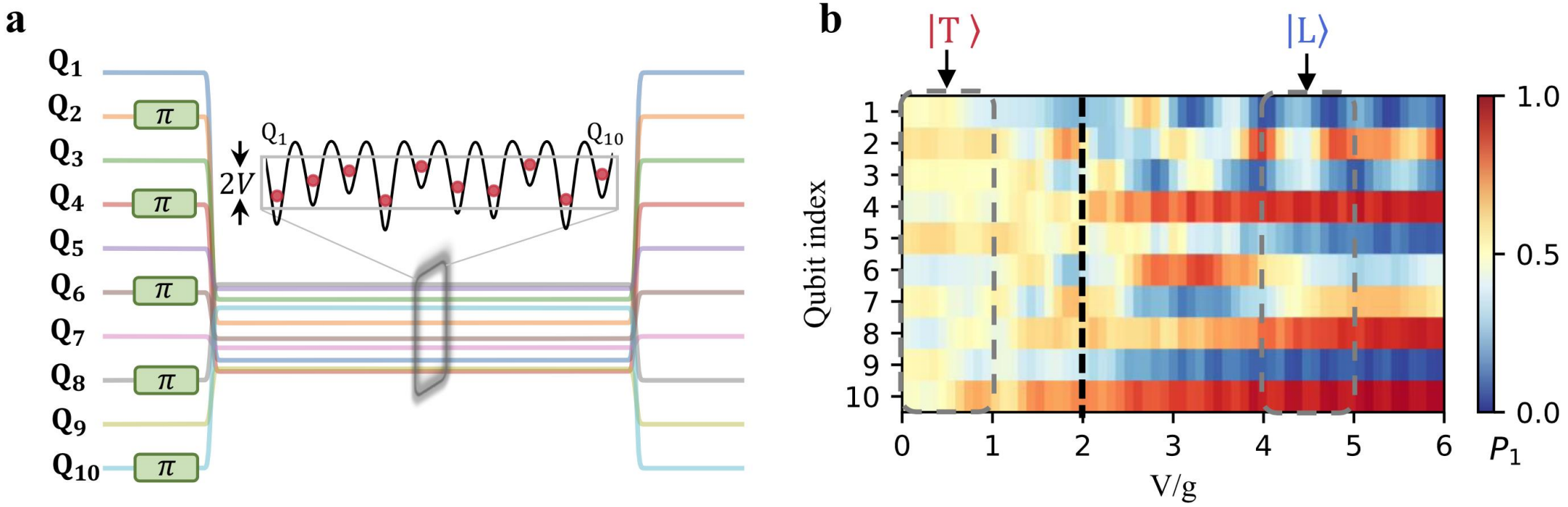
A phase transition at V/g=2:

- V/g>2: localized phase
- V/g<2: delocalized (thermal) phase

S. Aubry and G. André, Ann. Israel Phys. Soc 3, 18 (1980).

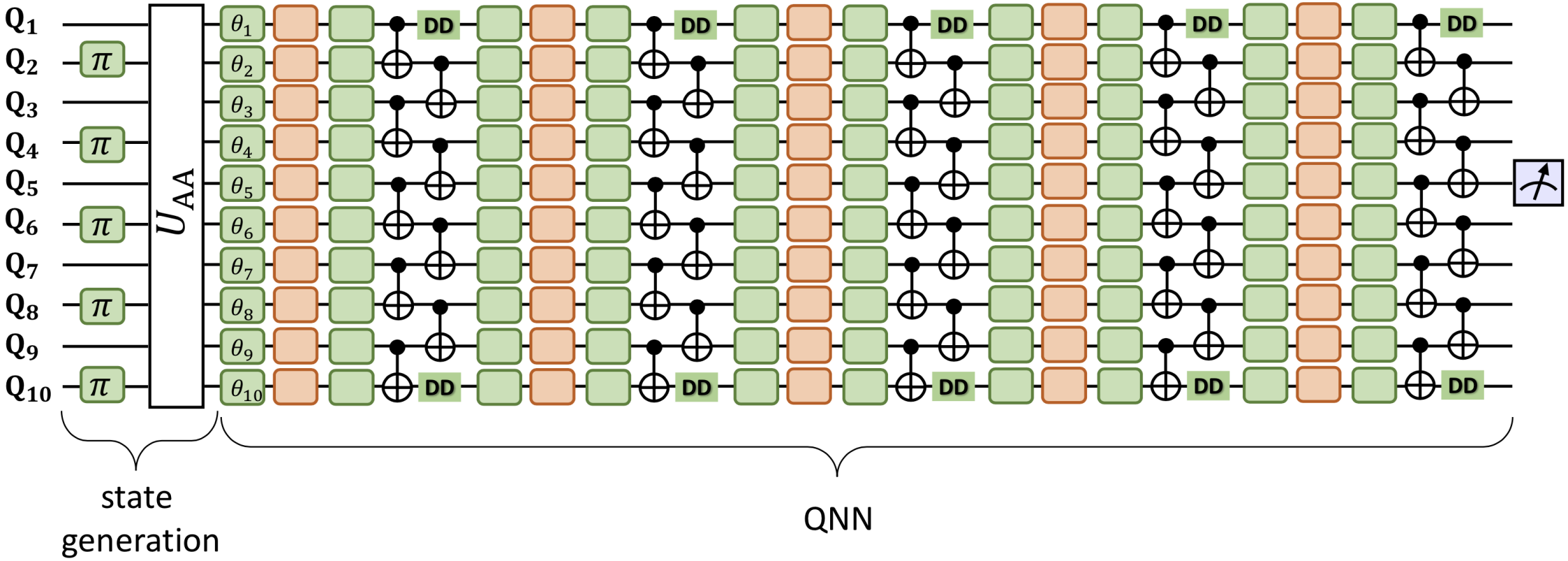
Model design & training

Experimental preparation for this quantum dataset:



Model design & training

Experimentally train a QNN with this quantum dataset:



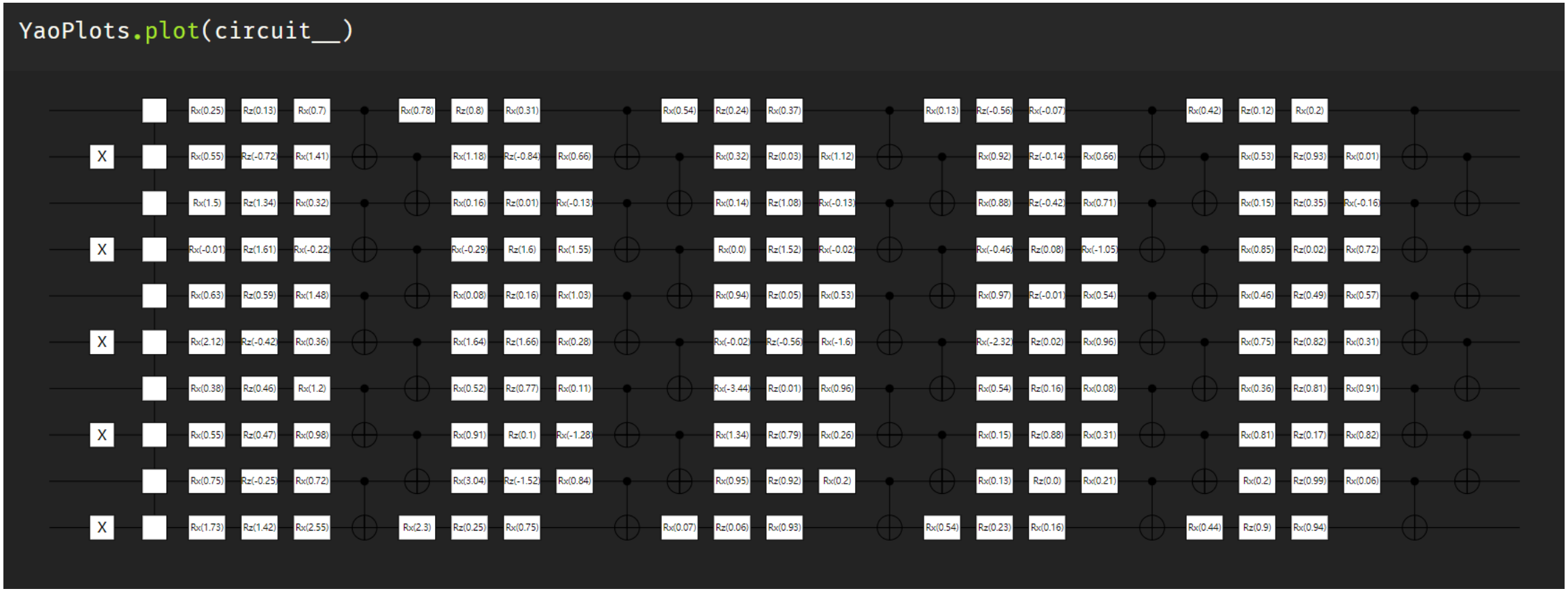
Model design & training

```julia
using Yao
using Plots
using Distributions
using SparseArrays
using LinearAlgebra
using MAT

function spin_operator(num_spins::Int, sites::Vector{Int}, opts::Vector{Int})
    ops = [sparse([0 1+0im; 1+0im 0]), sparse([0 -1im; 1im 0]),
        sparse([1+0im 0; 0 -1+0im]), sparse([1+0im 0; 0 1+0im]),
        sparse([0 1+0im; 0 0]), sparse([0 0; 1+0im 0])]

    idx = [4 for i = 1:num_spins]
    idx[sites] = opts

    opt = sparse([1])
    for i in 1:num_spins
        opt = kron(opt, ops[idx[i]])
    end
    return opt
end

function opts(num_qubits::Int)
    tz = [spin_operator(num_qubits, [k], [3]) for k = 1:num_qubits]
    txx = [spin_operator(num_qubits, [k,k+1], [1,1]) for k = 1:num_qubits-1]
    tyy = [spin_operator(num_qubits, [k,k+1], [2,2]) for k = 1:num_qubits-1]

    # next nearest neighbor terms
    pos_ = [[1,3],[4,6],[6,8],[7,9],[8,10]]
    txx_ = [spin_operator(num_qubits, k, [1,1]) for k in pos_]
    tyy_ = [spin_operator(num_qubits, k, [2,2]) for k in pos_]

    return tz, txx, tyy, txx_, tyy_
end
```

```julia
num_qubits = 10
num_exps = 1
α = (√5 - 1) / 2
ϕ = 0
# ϕ = rand(Uniform(0,2π))
hs = [cos(2π*α*k + ϕ) for k = 1:num_qubits]
# hs = hs + kkk
#==================num of qubits and random realizations============#

#==================operator============#
m = chain(num_qubits, [put(num_qubits-k+1=>X) for k = 2:2:num_qubits])
tz, txx, tyy, txx_, tyy_ = opts(num_qubits);
#==================operator============#

#==================time and disorder============#
t = 1*2pi

dm = 0
dw = 0.
gm = 1.
gw = 0.

dss = (dm .+ dw * rand(Uniform(0,1), num_exps)) * hs'
gss = gm .+ gw * rand(Uniform(-1,1), num_exps, num_qubits-1)
#==================time and disorder============#

states = zeros(ComplexF64, num_exps, 2^num_qubits)
u = []
p = []
@time for k = 1:num_exps
    h = (-sum(dss[k, :] .* tz)) .+ (-sum(gss[k, :] .* (txx + tyy))) .+ (0.2 .* sum(txx_ + tyy_))
    push!(u, exp(-im * t * h |> Array))
    push!(p, exp(-im * t * h |> Array) * (m |> Matrix) * (zero_state(num_qubits).state))
end

@const_gate u1 = u[1]
```

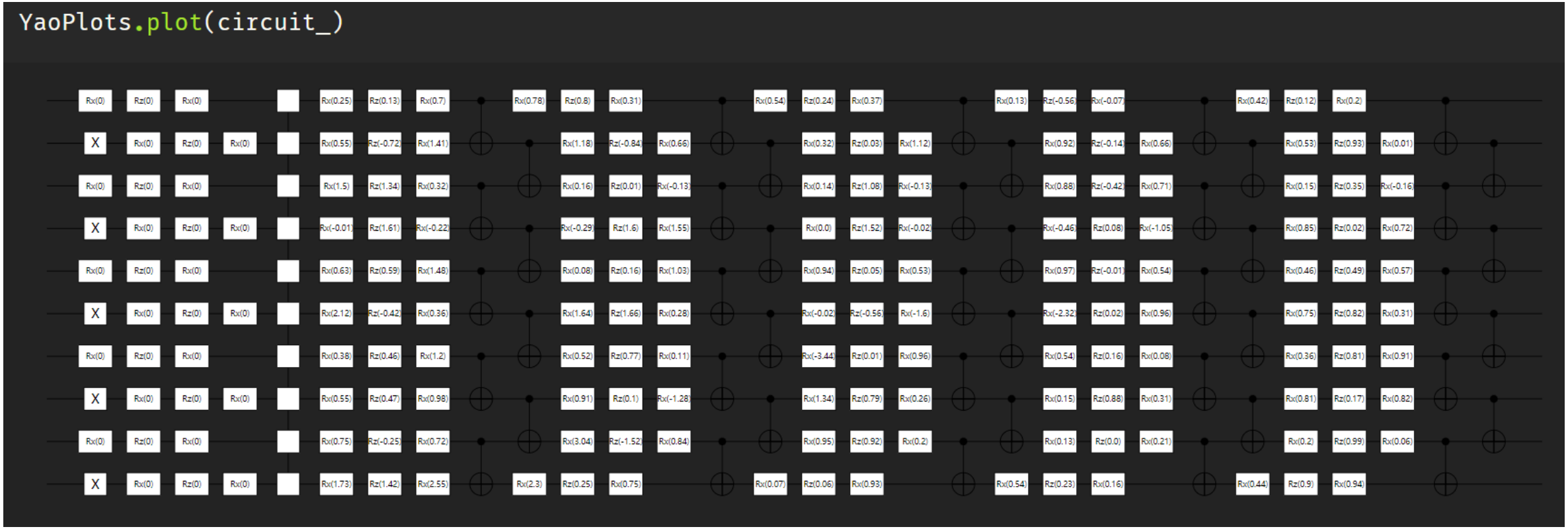
Model design & training

```
YaoPlots.plot(circuit__)
```



Model design & training

```
YaoPlots.plot(circuit_)
```



Model design & training

Define Custom Blocks

Primitive blocks are the most basic block to build a quantum circuit, if a primitive block has a certain structure, like containing tweakable parameters, it cannot be defined as a constant, thus create a new type by subtyping `PrimitiveBlock` is necessary

```
using YaoBlocks

mutable struct PhaseGate{T <: Real} <: PrimitiveBlock{1}
    theta::T
end
```

If your insterested block is a composition of other blocks, you should define a `CompositeBlock`, e.g

```
struct ChainBlock{N} <: CompositeBlock{N}
    blocks::Vector{AbstractBlock{N}}
end
```

Besides types, there are several interfaces you could define for a block, but don't worry, they should just error if it doesn't work.

Define the matrix

The matrix form of a block is the minimal requirement to make a custom block functional, defining it is super simple, e.g for phase gate:

```
mat(::Type{T}, gate::PhaseGate) where T = exp(T(im * gate.theta)) * Matrix{Complex{T}}(I, 2, 2)
```

Or for composite blocks, you could just calculate the matrix by call `mat` on its subblocks.

```
mat(::Type{T}, c::ChainBlock) where T = prod(x->mat(T, x), reverse(c.blocks))
```

The rest will just work, but might be slow since you didn't define any specification for this certain block.

Define how blocks are applied to registers

Although, having its matrix is already enough for applying a block to register, we could improve the performance or dispatch to other actions by overloading `apply!` interface, e.g we can use specialized instruction to make X gate (a builtin gate defined `@const_gate`) faster:

```
function apply!(r::ArrayReg, x::XGate)
    nactive(r) == 1 || throw(QubitMismatchError("register size $(nactive(r)) mismatch with block
    instruct!(matvec(r.state), Val(:X), (1, ))
    return r
end
```

In Yao, this interface allows us to provide more aggressive specialization on different patterns of quantum circuits to accelerate the simulation etc.

Define Parameters

If you want to use some member of the block to be parameters, you need to declare them explicitly

```
niparams(::Type{<:PhaseGate}) = 1
getiparams(x::PhaseGate) = x.theta
setiparams!(r::PhaseGate, param::Real) = (r.theta = param; r)
```

Define Adjoint

Since blocks are actually quantum operators, it makes sense to call their `adjoint` as well. We provide `Daggered` for general purpose, but some blocks may have more specific transformation rules for adjoints, e.g

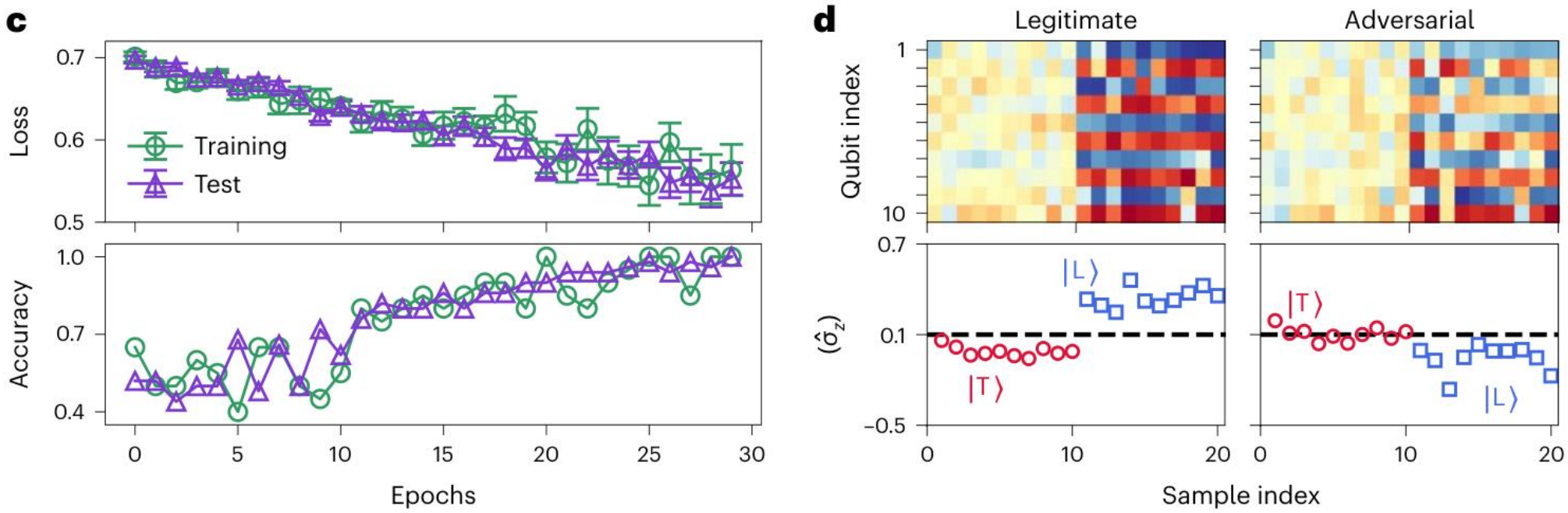
```
Base.adjoint(x::PhaseGate) = PhaseGate(-x.theta)
```

Define Cache Keys

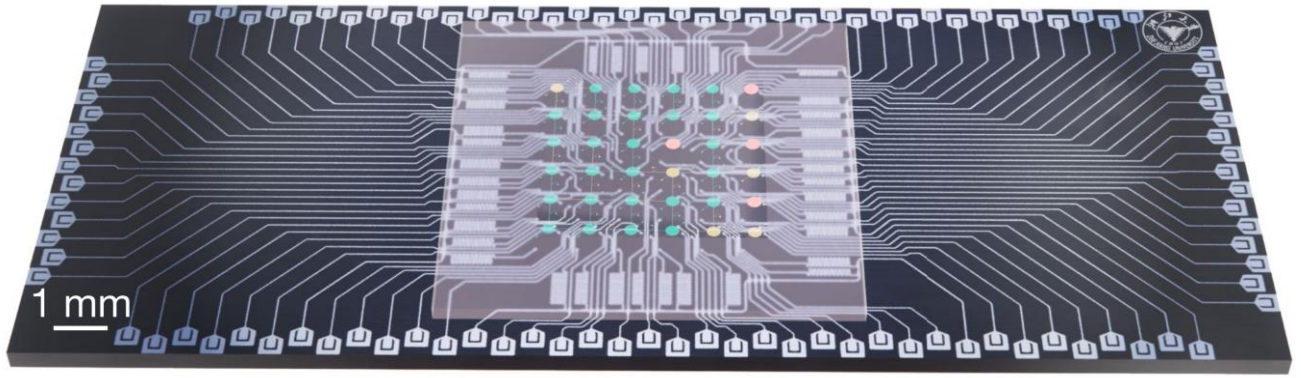
To enable cache, you should define `cache_key`, e.g for phase gate, we only cares about its phase, instead of the whole instance

```
cache_key(gate::PhaseGate) = gate.theta
```

Model design & training



Summary



- Experimentally demonstrated QAML with high-dimensional and real-life classical datasets / quantum many-body states, with Yao.jl providing a solid numerical support.
- Introduction to the code frameworks for this project.
- **Push the limit!**

Thank you for your listening!