UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Masters thesis

# Analyzing spin-qubit traces and dynamical quantum phase transitions using machine learning

Christian Bjørnholt

Advisor: Evert van Nieuwenburg

Submitted: December 17, 2021

# Analyzing spin-qubit traces and dynamical quantum phase transitions using machine learning

Christian Bjørnholt*

December 17, 2021

**Abstract**

This thesis investigates how machine learning techniques can be applied in the study of physics. The first topic is about improving the readout speed of semiconductor qubits by employing principal component analysis. This results in a one microsecond measurement time compared to the currently used method of homodyne detection that requires several microseconds. The time can be lowered even more by systematically discarding signals that yield unclear results. The second topic is about dynamical quantum phase transitions, stemming from non-equilibrium phase transitions that occur when a quantum state is evolved through time. They are studied through Loschmidt rate functions by finding their non-analytic points. By making recurrent neural networks evaluate these functions, it is possible to tell if they contain dynamical quantum phase transitions or not. More complex variations can even specify where they occur.

*chr.b@live.dk

# Contents

# 1 Introduction and Motivation

A lot physical systems are highly complex and it can therefore be hard to build them into a model. The usual workaround is to make assumptions about the system and and introduce approximations where an analytical solution is unfeasible. While this is good enough for many purposes, it can lead to a model that is missing important characteristics of the system. In light of this, we want to take a look at how machine learning could help us make deductions about these systems that we do not have a full overview of.

Machine learning is a set of algorithms that train a variable model from data towards a desired goal. It is a practice that has given rise to a lot progress many diverse areas, such as images (classification and generation [1]) and sentences (translation and emotion recognition [2]). Because machine learning has shown such promise in other areas, it is no surprise that it has begun to be used in physics as well. The goal of this thesis is to explore some of the potential use cases of machine learning can have in physics.

It starts off in Chapter 2 by introducing the concept of machine learning and how data can be used to train a model that produces something useful. The first type of machine learning we will be looking at is principal component analysis, which uses the variation in the provided data to produce an affine transformation that helps expose the features lying within the data. After that we talk about artificial neural networks which is a lot broader topic as they can take almost any form. The form we will be using is called recurrent neural networks, which are great at handling sequences of data with arbitrary length. However before we introduce those, we go through a one of the basic neural networks to understand how they work and how they are trained.

In Chapter 3 we look at what machine learning can do for quantum computers, specifically the classification of qubits when they are being measured. A qubit is a two-level quantum system that differs from a classical bit in that it can be in a superposition of its two states. When combined with more qubits, we get a new type of memory that allows one to perform algorithms that are impossible to do on classical computers. This has lead to algorithms designed specifically for quantum computers whose computational time scale better than their classical counterparts. As the qubits are genuine quantum states, they also serve as good candidates for performing quantum simulations [3]. We will be looking at a type of semiconductor qubit, which is measured using radiofrequency (RF) reflectometry. Normally the classification of a qubit is done through homodyne detection, but that can take several microseconds to do, which is far longer than the qubit coherence time. Our goal is to use principal component analysis to improve the qubit readout times while keeping a high fidelity.

In addition to this, we will also explore how recurrent neural networks can be applied to dynamical quantum phase transitions in Chapter 4. These are a type of non-equilibrium phase transitions that can occur by evolving a system in time. They are defined as points where the so-called Loschmidt rate function becomes non-analytic for an instant, which signify a drastic change in the time evolution operator [4]. We are going to train networks that can figure out if such a non-analytic point exists in the rate function, in which case a dynamical quantum phase transition occurred at some point.

At the end, in Chapter 5, we summarize the results of our findings and consider how our methods could be extended.

# 2 Machine Learning Methods

Before we begin using machine learning, we first need to understand what it is and how it works. Machine learning consists of two parts: a model and a learning algorithm. The model is simply something that produces an output in a desired format, which can for example be done by transforming a given input. While most algorithms have very fixed behaviour designed for a specific purpose by humans, a machine learning model consists of multiple numerical parameters that can be adjusted to entirely change its behaviour. This means it can be adapted to many different use cases with proper configuration. However in most cases it is be infeasible for a human to get any kind of interesting functionality as the set of parameters is either too vast or too sensitive to changes. That is why we need to train the model with another algorithm specifically designed for tuning the parameters according to some overall goal. This is the learning part of machine learning and is the reason why machine learning models are so flexible.

There are several ways to train a model, such as supervised learning and unsupervised learning, which are the ones we will be using. The goal of supervised learning is to generalize, so the model is given a set of training inputs accompanied by the desired outputs which it should be taught with. The learning algorithm will then try to minimize the difference between the target and model outputs until additional training yields negligible results. The model should then hopefully have been taught the important relations between the training inputs and outputs such that it can now predict outputs from entirely new inputs that are not part of the training data. On the other hand, unsupervised learning does away with any target outputs and only uses a set of inputs to teach with. The goal is to find the overall structure, which can be used to split inputs into different groups for classification or to generate data that is similar in nature to the training data.

Machine learning consists of many types of models, all with varying goals and design choices. In our case we are mostly interested in using some of these for classification, but that is just one of the many possible use cases of machine learning. The goal of a classification algorithm is to find how likely it is an input falls into one of several groups. Some classification models even produce the supposed probabilities of being in each group and therefore give very clear indication to how confident the model is in its answer.

## 2.1 Principal Component Analysis

Principal Component Analysis (PCA) is an unsupervised algorithm that can be understood in multiple ways [5]. In our case it will be used for classification, so we will view it as an algorithm that picks out the most prominent features of the dataset, the so-called principal components. It is then possible to compare the features of two samples in the dataset to conclude if they are different or not. To understand what exactly is meant by features, we need to discuss how PCA works, which is best done by introducing our dataset.

The dataset is a collection of $n$ samples $x_i$ with $i = 1, \dots, n$. Each sample is composed of $l$ variables $v_i$ with $i = 1, \dots, l$, such that they can be thought of as vectors living in the variable space $U = \mathbb{R}^l$. With this convention we can set $v_i = e_i$, such that the variables form a basis for $U$ through the standard basis $\{e_i \mid i = 1, \dots, l\}$. To shorten some expressions involving sample variance, we also want to center the dataset, which is easily achieved by translating each sample $x_i \to x_i - \bar{x}$, where $\bar{x} = \frac{1}{n}\sum_{j=1}^{n} x_j$ is the sample mean.

A very important concept of PCA is the sample variance of a variable. The variance of our current variables can be computed with

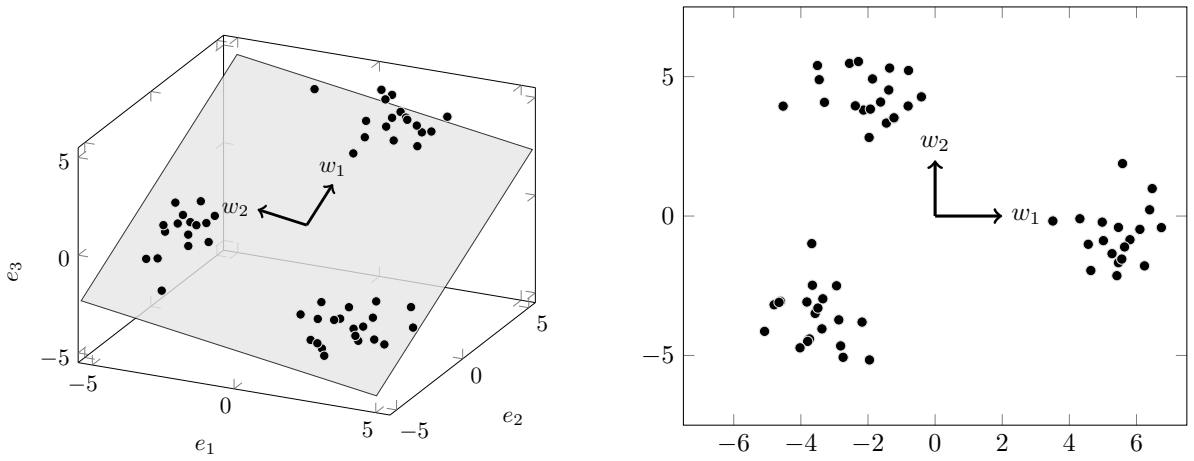
$$\mathrm{Var}(v_i) = \frac{1}{n-1}\sum_{j=1}^{n}((x_j)_i - \bar{x})^2 = \frac{1}{n-1}\sum_{j=1}^{n}(x_j)_i^2 = \frac{1}{n-1}\sum_{j=1}^{n}(x_j \cdot v_i)^2, \tag{1}$$

which together gives the total variance of the dataset $\mathrm{Var}(U) = \sum_{i=1}^{l}\mathrm{Var}(v_i)$. From the dot product $x_j \cdot v_i$ we see that it is actually possible to find the variance for any variable of $U$, meaning we do not have to restrict ourselves to the variables defined by our dataset. For example we could use another orthonormal basis $\{u_i \mid i = 1, \dots, l\}$ for $U$ and compute their variances. The total variance $\mathrm{Var}(U)$ will stay the same as a basis change can be thought of as a sequence of rotations. To express this point more rigorously, we will represent the dataset as an $n \times l$ matrix $X$ where the $i$'th row corresponds to $x_i$ and its $l$ variables. This lets us rewrite the variance for a variable $v$ as

$$\mathrm{Var}(v) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i \cdot v)^2 = \frac{1}{n-1}\sum_{i=1}^{n}(Xv)_i(Xv)_i = \frac{1}{n-1}(Xv)^{\mathrm{T}}(Xv) = v^{\mathrm{T}}\mathrm{Cov}(X)v, \tag{2}$$

where $\mathrm{Cov}(X) = \frac{1}{n-1} X^{\mathrm{T}} X$ is the sample covariance matrix of $X$ (need our earlier assumption $\bar{x} = 0$ to express $\mathrm{Cov}(X)$ this way). This shows us the total variance is actually the trace of $\mathrm{Cov}(X)$, which is known to be invariant under a change of basis.

Classifiable datasets usually form clusters of samples, but with more variables it can become hard to find these clusters. It would therefore be useful to know along which directions the clusters are spread apart in this high-dimensional variable space. One way to do this is to find a small set of new variables with high variances that can represent the directions the samples are spread out along. This is essentially what PCA is about. Specifically, the algorithm finds the subspace $V \subset U$ with the specified dimension $p < l$ that maximizes $\mathrm{Var}(V)$. We can then compute a basis for $V$, which forms a set of variables $\{w_i \mid i = 1, \ldots, p\}$ with high variances and gives us the relation $\mathrm{Var}(V) = \sum_{i=1}^{p} \mathrm{Var}(w_i)$ to work with. As an example, we can look at Figure 1 where the dataset uses three variables, but we want reduce that to two to get a better overview of the clusters. To do this, PCA forms the plane $V$ that the dataset gets projected onto, together with the variables $w_1$ and $w_2$ that form a two-dimensional coordinate system.




(a) (b)

Figure 1: **(a)** Three clusters with Gaussian distributions and a plane representing the two-dimensional subspace $V$ with the principal components $w_1$ and $w_2$. **(b)** The clusters projected onto $V$ with the principal components as the axes.

It turns out the best way to go about finding $V$ is to diagonalize $\mathrm{Cov}(X)$ and get its eigenvectors $\{u_i \mid i = 1, \ldots, l\}$ and their respective eigenvalues $\lambda_i$. Then in Apx. A we show if the eigenvectors are ordered by highest eigenvalue ($\lambda_1 \geq \lambda_2 \geq \ldots$), then $V = \mathrm{span}\{u_i \mid i = 1, \ldots, p\}$ and we get a new set of variables by setting $w_i = u_i$. These variables are called the principal components and are used to project the samples onto $V$ (by using inner products for example). Because $\mathrm{Var}(w_i) = \lambda_i$, we find $w_1$ is the variable with the highest variance in $U$, and if it was removed, $w_2$ would become the variable with the highest variance. Repeating this argument for $i > 2$, the new set of variables can be said to be ordered by the highest possible variance they can have, making them an ideal choice without doing any extra case-by-case analysis.

The last question is if there is some systematic way to choose how many dimensions $V$ should have, as unlike Figure 1, we have no good way to visualize the projection onto $V$ for $l > p > 3$. This is where the concept of explained variance comes in [6], which is simply the fraction of variance each principal component conserves

$$f_i = \frac{\mathrm{Var}(w_i)}{\mathrm{Var}(U)} = \frac{\lambda_i}{\sum_{j=1}^{l} \lambda_j}. \tag{3}$$

The strategy is to choose the fractions $f_i$ that dominate over the rest in value, such that only variables with a high variance (relative to the rest) will be used. This approach works because at some point we expect to run out of variables that span the clusters in the dataset, meaning the variance of the remaining variables would come from noise. This of course assumes the noise in the dataset is lower than the spread of the clusters, as it would otherwise be difficult to know if a variable represents a useful feature instead of noise. As an example, in Figure 1 the fractions are $f_1 = 0.578$, $f_2 = 0.400$, and $f_3 = 0.022$, so using this method we would choose $p = 2$, as the value of $f_3$ is much lower than the other two fractions and it could be considered noise.

## 2.2 Neural Networks

A neural network is a model that gains inspiration from how neurons in a biological brain are wired together. One of the simplest versions is the feedforward neural network (FNN), which we will use to introduce neural networks. Its general structure has been sketched in Figure 2 and can be seen to consist of nodes and directed edges. The nodes are meant to mimic the neurons of our brain and the edges are their numerous connections. An important observation is that a FNN actually consists of layers of nodes and the edges only go from one layer to the next. For example Figure 2 contains three layers, each respectively consisting of three, five, and two nodes going from left to right.

The layered structure means the model is executed sequentially by treating the output of a layer as the input for the next layer. In practice this is done by treating the output of a single layer as a vector with a dimension matching the number of nodes. We can therefore write the output of the $i$'th layer as the vector $x_i$. From here, we can express the output of the next layer $i+1$ using an affine transformation together with a non-linear function $f_{i+1}$

$$x_{i+1} = f_{i+1}(W_{i+1}x_i + b_{i+1}). \tag{4}$$

The affine transformation contains the so-called weights $W_{i+1}$ and bias $b_{i+1}$ for layer $i+1$. In a FNN, it is these two terms that can be varied to modify the behaviour of the network. The matrix $W_{i+1}$ can control how the layer interacts with the previous layer, but because $f_{i+1}$ is a fixed function, we may also need to use the vector $b_{i+1}$ to increase the flexibility of the network. The non-linear function usually returns a vector with the same dimensions as its input, so it is possible to specify the dimension of every object beforehand. Given $d_i$ is the dimension of $x_i$, we therefore have the following definitions

$$x_i \in \mathbb{R}^{d_i}, \quad f_i : \mathbb{R}^{d_i} \mapsto \mathbb{R}^{d_i}, \quad W_i \in \mathcal{M}(d_i, d_{i-1}), \quad b_i \in \mathbb{R}^{d_i}, \tag{5}$$

where $\mathcal{M}(m, n)$ is the set of $m \times n$ real matrices.

The final part of a FNN is the starting input and the final output. In Figure 2 we have three layers, the input layer, a middle layer, and the output layer. The output of the input layer is simply the input given to the FNN, so $x_1$ equals the input in a vectorized form. The middle layer is a so-called hidden layer, which just means it is neither an input nor an output. In this example there is only one hidden layer, but the number increases by one with each additional layer that is added. Finally we get the actual output of the FNN from the final layer, which in this example would be $x_3$. It is of course possible to manipulate this output further, but that would not be part of the FNN.

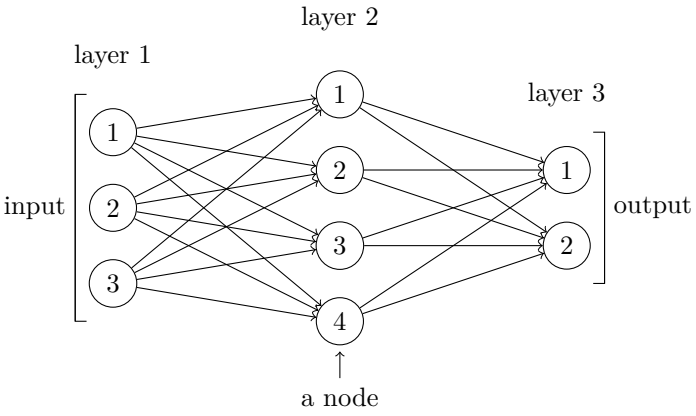



Figure 2: A small FNN with three layers. Each layer contain nodes, which are meant to look like biological neurons when connected. The first layer contains all the input information that the FNN will use. The second layer is only used to give the FNN flexibility. The result of last layer becomes the output of the FNN.

### 2.2.1 Training

We can now give a neural network an input and make it produce an output, but it is currently not the output we desire. This is because the weights and biases need to be configured properly to give something sensible, and for that we need to train the neural network. As mentioned earlier, there are more than one way to train a neural network, but we will only focus on supervised learning as it is straightforward to

use in our case. The distinctive part of supervised learning compared to other methods is that for every input $x$ in the training data, there is an accompanying target output $y$. The goal of the neural network during its training is to reproduce these target outputs when given the respective training inputs.

Now that we understand the idea behind supervised learning, we can begin to explain how it is done in practice. To start with, we need to know how well the neural network matches the target outputs. This leads us directly to loss functions, which are real functions with the main property of reaching their global minima when two quantities become equal. In our case these quantities would be the neural network output $\mathrm{NN}(x)$ and the target output $y$. As an example, given the single input $x$, if the output vector $\mathrm{FNN}(x)$ of a FNN should match the target vector $y$, then we could apply the mean squared error loss function $L(x) = \sum_j (\mathrm{FNN}(x)_j - y_j)^2$ and begin minimizing it by changing the weights and biases.

This brings up the question of how we are going to minimize loss functions. The answer is gradient descent, an algorithm for finding local minima of a multivariable function. Because we are going to manipulate the weights and biases, we will expose them in the notation for the loss function $L_\theta(x)$ where $\theta = \{W_i, b_i\}$. With this we can write down the gradient $\nabla_\theta L_\theta(x)$ of the loss function evaluated at the current $\theta$ of the neural network. The gradient can be calculated individually for each weight and bias by viewing the neural network as a big concatenation of functions and applying the chain rule. This method of calculating the gradient is called backpropagation and an in-depth derivation for the FNN can be found in [7].

We are now able to do gradient descent by updating the weights and biases into $\theta \to \theta - \eta \nabla_\theta L_\theta(x)$ as illustrated in Figure 3. Basic gradient descent comes with the learning rate $\eta > 0$ which controls how much the weights and biases should change with each iteration of gradient descent. With a small enough $\eta$ it can be shown this change to $\theta$ does actually lower the value of the loss function, which is the reason gradient descent works and can find local minima through iteration. Right now $\eta$ is just a constant, but with a more sophisticated learning algorithm, such as the Adam optimization algorithm [8] that we will be using, it can be automatically adjusted for each iteration. This can be important as $\eta$ should be lower around local minima to avoid overshooting them with a big change to the weights and biases. On the other hand, a low $\eta$ when the loss function is not near a local minima can slow the learning process down as less progress is made with small step sizes.

This is all for a single input in the training dataset, but to properly teach the neural network we have to include the whole training dataset in some form. The most straightforward way would be to calculate the average loss of all inputs

$$\bar{L}_\theta = \frac{1}{n} \sum_{i=1}^{n} L_\theta(x_i) \tag{6}$$

and perform gradient descent on that. While this works, it is not very efficient as the neural network has to evaluate all inputs every iteration. To get around this we can split the training dataset into multiple smaller datasets called batches. We can then choose a single batch that will be used for gradient descent, which will be much faster now that the number of inputs has been lowered. Then by performing gradient descent on each batch in succession, we go through multiple iterations, while still taking all available training data into account. In most cases one run-through of the training dataset does not yield enough iterations to properly teach the neural network. We therefore need to go though the training dataset multiple times, with one full run-though being called an epoch. The training dataset should also be shuffled after each epoch when creating the batches to avoid possible learnable patterns.

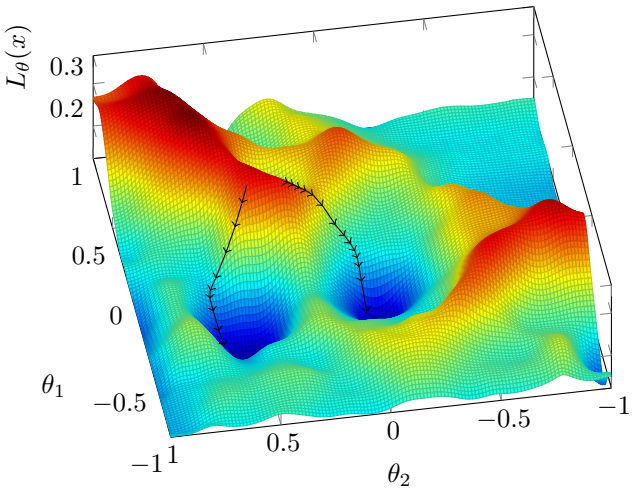



Figure 3: A loss function $L_\theta$ with the two variables $\theta_1$ and $\theta_2$ is given the input $x$. Two different initial values of $(\theta_1, \theta_2)$ were used, where each converges towards a different minima with the learning rate $\eta = 0.85$.

### 2.2.2 Classifiers

The FNN is just one of many possible neural networks, so it is a given to choose one that better fit the requirements for the task at hand. In our case, we are interested in using neural networks for classification, which does not have a neural network type that generally outperforms other types. We can, however, make a small modification to existing neural networks and make them more suitable for divide a dataset into classes. Classification is all about making the best choice, but we can also think of this as selecting the most probable class out of many. Extending this idea further, if we can make the model assign a probability to each class, which together forms a probability distribution, we get an insight into how confident the model is about each class.

Doing this in practice is actually very simple as long as the output dimension of the neural network matches the number of classes. If it does, we are able to add the softmax function

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{7}$$

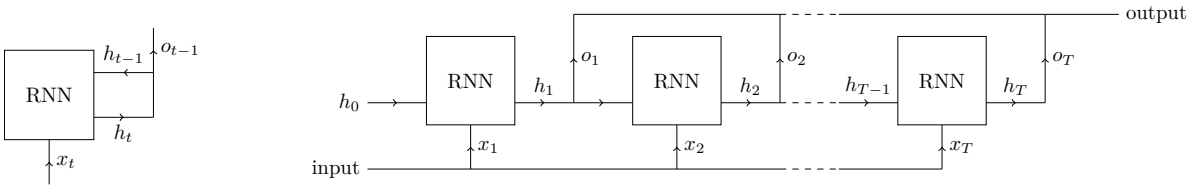
at the end of the neural network. We see the softmax gives us a probability distribution since $0 < \text{softmax}(x)_i < 1$ and $\sum_i \text{softmax}(x)_i = 1$, so it is a perfect candidate for classifying using probabilities.

Behind the scenes, the neural network has no idea what a probability distribution is and just tries scale the entries of vector given to the softmax function such that it would match the target output during training. It is therefore not a given that the neural network returns a sane probability distribution as it is only us that actually know what probabilities are.

## 2.3 Recurrent Neural Networks

While a FNN is one of the simplest models that can be used, the number of columns of the first weight matrix $W_1$ scales linearly with the number of input parameters. This might hurt the interpretability of the neural network if the entries in $W_1$ act very differently on every input, making it impossible to understand how everything plays together. This would especially be true for inputs that can be viewed as a sequence of samples. Here we would intuitively expect the neural network to treat every element of the sequence the same way, as they do not contain any distinguishing features by themselves. It is only when they are put together into a sequence that we would see any interesting characteristics.

We can try to solve these problems with another type of neural network. We are going to look at recurrent neural networks (RNN), which is specifically designed to handle sequences of data. An RNN functions by going through each point in a sequence one by one in order. It treats every point the same, meaning only one set of weights and biases are used across the whole sequence. However, it still needs to somehow take multiple points into consideration, as no single point would hold any useful information as we discussed. An RNN therefore also has a built-in memory that gets passed along from one point to the next as the RNN goes through the sequence.




(a) (b)

Figure 4: **(a)** A sketch of an RNN that uses the input $x_t$ and the previous hidden state $h_{t-1}$ to produce a new hidden state $h_t$. This hidden state then loops back into the RNN to join up with the next input, while also being the output ($o_{t-1}$) of the RNN at this timestep. This procedure is repeated for every element in the input sequence. **(b)** The unravelled version of an RNN, where the whole sequence is laid out. Just like the input is a sequence, the hidden states also form an sequence that is used as the RNN output.

There are even multiple types of RNNs, so we will start by discussing the simplest one. We will mark the position of a point in the sequence with a time $t$, such that the $t$'th input would be the vector $x_t$. At each time $t$ the RNN will also modify its memory by generating the vector $h_t$, which is formally called the hidden state. The way the hidden state is computed at each timestep is very similar to a FNN with its weights and biases

$$h_t = f(W_i x_t + W_h h_{t-1} + b). \tag{8}$$

We see that an RNN uses the two weights $W_i$ and $W_h$, together with the bias $b$ and a non-linear function $f$. The weight that interacts with the inputs is $W_i$, and works just like it would in a FNN. On the other hand, $W_h$ uses the previous hidden state $h_{t-1}$ to alter the behaviour of the RNN at time $t$, which is why the hidden states can be seen as the memory of the RNN. Just like for the FNN, we specify

$$x_t \in \mathbb{R}^{d_i}, \quad h_t \in \mathbb{R}^{d_h}, \quad f : \mathbb{R}^{d_h} \mapsto \mathbb{R}^{d_h}, \quad W_i \in \mathcal{M}(d_h, d_i), \quad W_h \in \mathcal{M}(d_h, d_h), \quad b \in \mathbb{R}^{d_h}. \tag{9}$$

The hidden states also serve as the output of the RNN. As they are generated at each timestep, the hidden states form a new sequence from the input sequence. This could be useful for simple machine translations, where the input sequence is a sentence and the hidden states would form a sequence of words in a different language that should hopefully be a legible sentence. For classification there is not a need for a sequence as an output, so it would be a better idea to just choose the last hidden state of the RNN, which could be thought of as its final answer. However this does not mean the sequence of hidden states has no use in classification. Because it can be seen as a sequence, it means another RNN can use it as its input, thus allowing us to layer RNNs, just like how a FNN consists of multiple layers of neurons. As the new RNN also generates hidden states, we still get the same type of output. This layering approach allows for more complex models while still keeping the core aspects of an RNN.

This RNN model is one of the most basic RNNs, so it does not perform as well at some tasks as other variants do. One of its downsides is its inability to retain its memory through the hidden state because it only uses a single variable affine transformation to control how it remembers. Another flaw is not so much in the model itself, but rather how gradients are computed using the chain rule. Because an RNN goes through multiple timesteps, we find that these timesteps will appear in the chain rule. As every timestep includes multiplications with its weights, it means the gradient acquires terms proportional to $w^T$, where $w$ is an entry of either $W_i$ or $W_h$ and $T$ is the sequence length. This exponential can either become minuscule or grow very large, depending on if $|w| < 1$ or not. This is called the vanishing or the exploding gradient problem and there is no good way to fix it for a basic RNN other than attempting to clamp the gradient value.

Luckily there is the popular long short-term memory (LSTM) RNN, which partially solves both of these problems. This also means it is a lot more complicated and uses more weights and biases compared to the basic RNN that we just discussed. The LSTM uses the following model

$$i_t = f_i(W_{ii} x_t + W_{ih} h_{t-1} + b_i), \tag{10a}$$
$$f_t = f_f(W_{fi} x_t + W_{fh} h_{t-1} + b_f), \tag{10b}$$
$$g_t = f_g(W_{gi} x_t + W_{gh} h_{t-1} + b_g), \tag{10c}$$
$$o_t = f_o(W_{oi} x_t + W_{oh} h_{t-1} + b_o), \tag{10d}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ g_t, \tag{10e}$$
$$h_t = o_t \circ f_h(c_t), \tag{10f}$$

where $\circ$ is the Hadamard product that takes the element-wise product of two vectors. Everything in the equations follows the same types of definitions as Eq. 9. The first four vectors $i_t$, $f_t$, $g_t$, and $o_t$ are called gates, while $c_t$ is the cell state and $h_t$ is the hidden state. In an LSTM both $c_t$ and $h_t$ are used as its memory, while the gates are used to control it, which we can intuitively explain through the equations. The gate $f_t$ is called the forget gate and it controls how much the LSTM should forget about previous timesteps. This behaviour can be seen in Eq. 10e, where a small $f_t$ would result in almost nothing of $c_{t-1}$ carrying over into $c_t$. On the other hand, we have the input gate $i_t$ and the store gate $g_t$, where $g_t$ is what the current memory should be replaced with and $i_t$ adjusts how much it should be replaced by. Last is the output gate $o_t$ seen in Eq. 10f, which simply allows for a greater degree of control over the hidden state that is used as the LSTM output.

The more complex memory model of an LSTM solves the memory problem of the simple RNN by giving the model direct control over what should be remembered and forgotten. An LSTM also improves upon the vanishing and the exploding gradient problem, by having $f_t$ included at every timestep in the gradient. Because $f_t$ can vary along the sequence, it avoids giving an exponential term in the gradient and thus partially solves the problem.

# 3 Classifying Qubits

In this chapter we will be looking at the possibility of improving the readout time of semiconductor spin-qubits. The qubits we will be looking at currently use homodyne detection for classification, which is a technique used in radio technologies. However, demodulating the qubit readout signal using this method is not ideal as it can take several microseconds for the readout to complete. These timescales are far higher than those used for manipulating the qubits and can lead to qubits losing information through decoherence. We therefore want to replace homodyne detection with a machine learning technique that uses the raw output signal to try improve readout times and, in effect, reduce decoherence. The algorithm we are going to use is principal component analysis (Section 2.1), which allows us to cluster a batch of output signals without needing complete prior knowledge of what kind outputs the experimental setup produces. PCA is also very fast to execute as it only uses an affine transformation. We are hoping PCA can be put on a field-programmable gate array (FPGA), so it can be implemented as part of the hardware of the experimental setup. This would improve the scalability of the readout setup when measuring multiple qubits as the number of components needed for homodyne detection increases with the number of qubits.

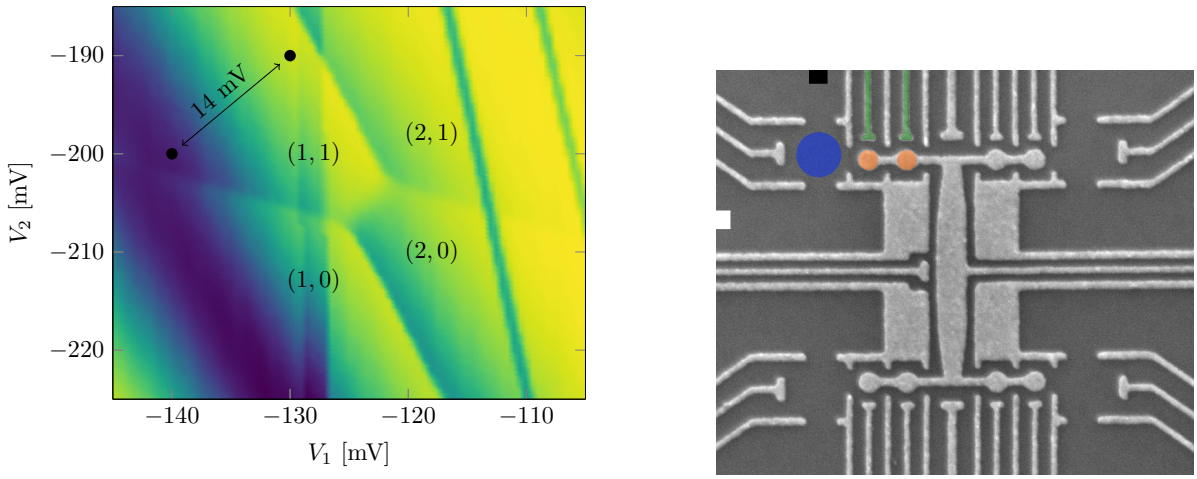
## 3.1 Double Quantum Dot Qubit

The experimental setup we are going to be working with uses a double quantum dot (DQD) to create a qubit. A quantum dot can be thought of as an artificial atom, which allows one to trap one or more electrons in place by adjusting a voltage gate. So with a DQD we get a two-dimensional voltage space that can be separated into sections by the amount of electrons residing in each quantum dot, leading to the charge stability diagram (see Figure 5a). By operating the DQD with electron occupations $(1,1)$ and $(2,0)$ we are able to create a singlet-triplet qubit by manipulating the singlet-triplet spin states $\{|S\rangle, |T_0\rangle\} = \{(|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle)/\sqrt{2}, (|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle)/\sqrt{2}\}$. If we first consider the DQD to be in the $(2,0)$ configuration, we find the exchange interaction $J$ between the electrons is the dominant factor because of the Pauli exclusion principle, making $|S\rangle$ and $|T_0\rangle$ the eigenstates of the DQD. On the other hand, in the $(1,1)$ configuration, the exchange interaction will disappear because of the distance between the quantum dots. As a consequence, we get the degenerate ground states $|\uparrow\downarrow\rangle$ and $|\downarrow\uparrow\rangle$. This degeneracy can be mitigated by giving the quantum dots parallel magnetic fields with a difference in strength of $\Delta B_{\parallel}$ to vary the Zeeman splittings. This leads us to the Hamiltonian [9]

$$H = \frac{J}{2}\sigma_z + \frac{\Delta B_{\parallel}}{2}\sigma_x, \tag{11}$$

where control over $J$ and $\Delta B_{\parallel}$ lets us reach any state on the Bloch sphere, thus achieving an operable qubit.

To detect what state the qubit is in, we use the fact that only the singlet state can exist while the system is in the $(2,0)$ configuration because it upholds the Pauli exclusion principle with its anti-symmetry. Using this, the method of distinguishing the singlet and triplet states starts by already having the DQD in the $(1,1)$ configuration, whereafter we will change $J$ such that the DQD favors $(2,0)$. If the qubit is in the $(1,1)$ singlet state, one of the electrons can jump to the other quantum dot and form the $(2,0)$ singlet state instead. On the other hand, if it is in the $(1,1)$ triplet state, it will be unable to change to the $(2,0)$ singlet state as that would violate spin conservation [10]. Together this means there will be a measurable difference in charge on both quantum dots depending on what state the qubit was in. To convert this to a signal we use a sensor quantum dot (SQD) which changes its resistance based on the surrounding charge. We can then send an radiofrequency signal into the device containing the SQD, which produces a reflected signal that constitutes the raw output signal that we will be using.




(a) (b)

Figure 5: **(a)** The charge stability diagram that is used to determine the electron occupation in each quantum dot with respect to the gate voltages $V_1$ and $V_2$. The visible regions have clear lines of separation and are labeled with the number of electrons in each quantum dot. The two dots are measurement points with a distance of 14 mV. **(b)** Image of the four qubits contained in the FF1A device from the Center for Quantum Devices at the Niels Bohr Institute. On the top left a pair of quantum dots (orange) is highlighted together with their connections (green) to the voltage gates. The SQD (blue) is located right next to the DQD in the electron electron reservoir (dark grey). The RF signal is sent from the white square, which continues through the SQD to the black square where the signal gets grounded. The reflected signal exits through the white square again, which is then sampled and used for classification.

## 3.2 Classifying Experimental Signals

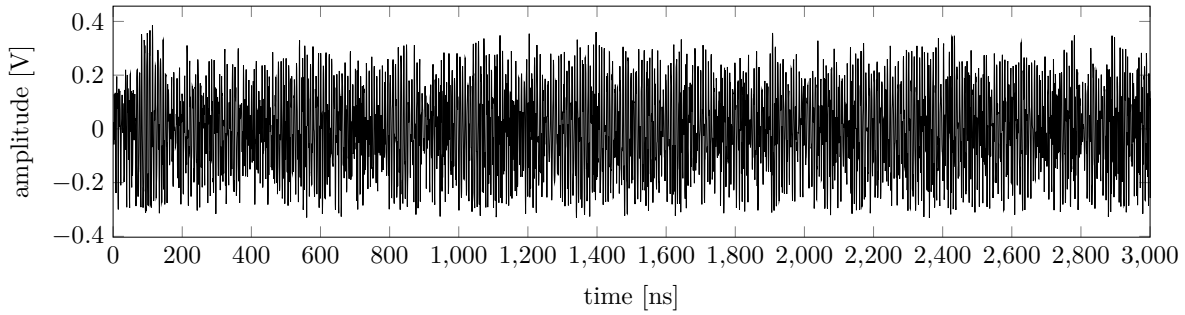



Figure 6: An example of a signal that will be classified.

Now that we know a bit about the experimental setup, we can begin analysing its raw output signals. While the idea is to classify the output of a singlet-triplet qubit, we did not get hold of such data. Instead we have outputs from a couple of states at different points in the $(1, 1)$ section of the charge stability diagram. This should still give us a good idea of what machine learning techniques are capable of, so we can extend its use to actual singlet-triplet qubits without worry.

The digitized signals we will be analysing are sampled for three microseconds with a sampling rate of 1.8 GHz. An example of such a signal can be seen in Figure 6. To follow the plan of using PCA, we need to have a vector space. This is done by viewing the signals as a vector of voltages, where each entry corresponds to one of the samples, which are preferably ordered by ascending time. As we are interested in classifying a single qubit for now, we should only need one principal component. Assuming each signal contains $T$ samples, we get a principal component vector $w$ with the same dimension after using PCA on the selected dataset.

Beginning our analysis, we choose two states with a 14 mV separation in the charge stability diagram (see Figure 5a). Each state is recorded 1000 times and to make sure the model is able to generalize, we only use 20% of the recorded signals for the PCA. As an initial test, we use all three microseconds,

which yields a perfect splitting of the dataset with the single principal component, as seen in Figure 7. So PCA can achieve 100% accuracy for signals that run for microseconds, however when we begin to use less and less of the signal we see a noticeable decrease in accuracy which Figure 8a illustrates. Only using up to 300 ns gives no separation in the dataset, but right after that the accuracy begins to shoot up until it straightens out around 1000 ns with an accuracy approaching 100%.

In the case we want to push for sub-microsecond readout times, it is possible to discard signals that we are unsure about. As the classification errors stem from the two clusters overlapping around zero, we can select only those with a high enough absolute PCA transformation value to ensure we do not use any signals in the overlapping area. Figure 8b shows an example of this approach for 500 ns signals. As expected, the accuracy rises when we impose a higher minimum absolute value for the transformed signals, but in return the number of surviving signals also decrease. It therefore becomes a balancing act that depends on the desired readout time, accuracy, and fraction that is left of the recorded data.

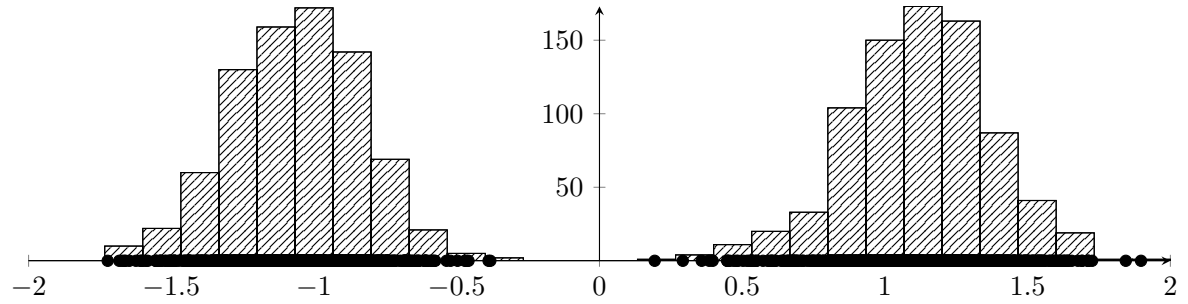
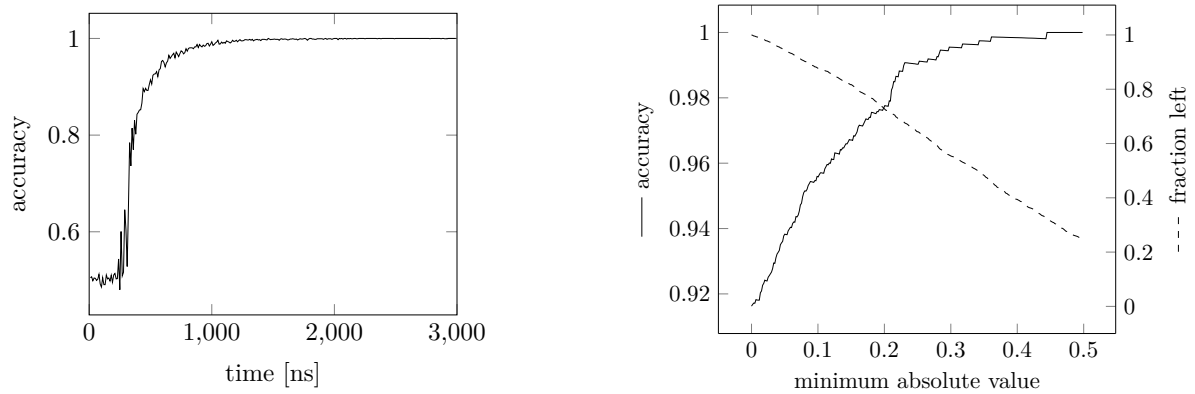



Figure 7: The signals after having gone through the PCA transformation. The dots on the horizontal axis show the projected values of the output signals. All signals lie comfortably on one of the sides of the horizontal axis, conveying all signals have been classified correctly. The two histograms formed by the projected values show the density of the clusters that lie underneath.




(a) (b)

Figure 8: **(a)** The classification accuracy as a function of the amount of time used. The accuracy starts at 50%, meaning the classification did not work. It then quickly begins to rise and later flattens out at around 1000 ns towards a perfect classification. **(b)** The accuracy of 500 ns signals when discarding ambiguous signals whose transformed values are too close to zero. The horizontal axis is signifies the minimum absolute value the transformed signals should have to be kept during classification. The solid line, together with the left vertical axis, shows the accuracy rising as more and more signals are excluded. The dashed line and the right vertical axis show the fraction of signals left for classification dropping as the requirements to remain become stricter.

### 3.2.1 PCA and Homodyne Detection

The PCA approach for classifying qubits looks promising, but we still do not know how the splitting works. However before we get to that, it is constructive to first discuss how homodyne detection is done. The input RF signal that is sent into the device has a single frequency near one of the resonant frequencies of the device, e.g. $s(t) = A_s \cos(\omega t)$. This gives rise to a reflected signal (the output), which

shares the same frequency, but with an amplitude and phase that changes depending on the resistance of the sensor dot, e.g. $r(t) \sim A_r \cos(\omega t + \theta_r)$. This means it is possible to distinguish the states by figuring out the amplitude $A_r$ and the phase $\theta_r$ of the reflected signal as those are different for each state. This is especially apparent if we try averaging over the measurements from each state, which has been done in Figure 9 and shows us a clear gap between the two amplitudes.

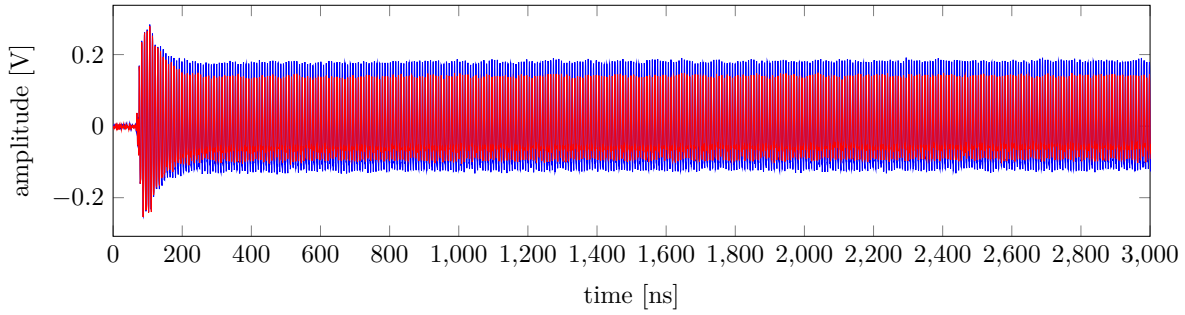



Figure 9: The output signal averages of both states colored red or blue. The averaged signals share the same amplitude at the start, but after that the blue colored average stays above and below the red one until the end.

Homodyne detection retrieves the amplitude and phase by first splitting $r(t)$ into two and applying a $\pi/2$ phase shift to one of them [11], giving us

$$r_I(t) \sim \frac{A_r}{2} \cos(\omega t + \theta_r) \quad \text{and} \quad r_Q = \frac{A_r}{2} \sin(\omega t + \theta_r). \tag{12}$$

From here we can average over the products $I(t) = \frac{1}{2} s(t) r_I(t)$ and $Q(t) = \frac{1}{2} s(t) r_Q(t)$ to get

$$I = \frac{1}{T} \int_0^T I(t)\,\mathrm{d}t = \frac{A_s A_r}{8} \cos(\theta_r) \quad \text{and} \quad Q = \frac{1}{T} \int_0^T Q(t)\,\mathrm{d}t = \frac{A_s A_r}{8} \sin(\theta_r), \tag{13}$$

where we assume $\omega \gg 0$ to do away with $\omega^{-1}$ terms. With this we are able to estimate

$$\frac{A_s A_r}{8} = \sqrt{I^2 + Q^2} \quad \text{and} \quad \tan(\theta_r) = Q/I, \tag{14}$$

which gives us the necessary values to make a distinction between states. It can also be viewed as a point in the complex plane $I + iQ = \frac{A_s A_r}{8} e^{i\theta_r}$ that the experimental results would cluster around.

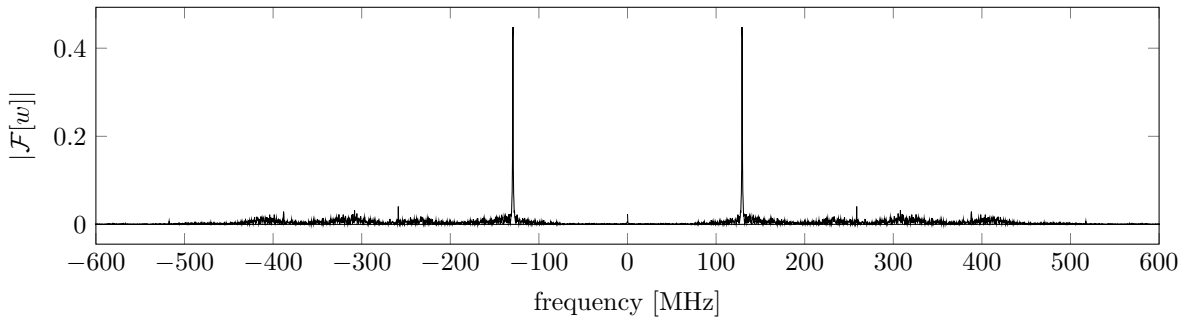



Figure 10: The Fourier transform of the principal component $w$ where the absolute value was taken at each frequency. The plot has a distinctive peak at 130 MHz, which matches the frequency of the output signals. The principal component can be understood as a single-frequency oscillation from how its peak dominates over all other frequencies.

If we now turn to PCA, it will become evident that the transformation it does is comparable to homodyne detection. Writing it out, the transformation is $(x - \bar{x}) \cdot w = \sum_{i=1}^{T} w_i x_i - \bar{x} \cdot w$, where $x$ is the output signal we want to transform and $\bar{x}$ is the mean of all output signals. The sum term bears a clear resemblance to $(I, Q) = T^{-1} \int_0^T \frac{1}{2} s(t) r_{(I,Q)}(t)\,\mathrm{d}t$ where $s(t)$ could easily be replaced by $w$, yielding

a result similar to the PCA transformation without the fixed offset created by $\bar{x}$. By taking the Fourier transform of $w$, plotted in Figure 10, we even see it is a single frequency wave that matches the frequency of the reflected signal of around 130 MHz. So PCA essentially recreates homodyne detection, but where $s(t)$ can have a phase that maximizes the separation such that we do not need to extract the phase to classify the output signals. This behaviour is even more useful in scenarios where homodyne detection has only been partially implemented in a way where the phase is not found. As a consequence, the reflected signal has to go through a manually calibrated phase shift to get the most information from the amplitude alone. PCA more or less automates this phase shift calibration, removing all trial and error needed from the process.

### 3.3 Simulating Multi-Qubit Signals

Using PCA allows us to classify a single qubit, but what about multi-qubit outputs? We already know PCA can handle multiple features through its principal components, so all we require is a dataset containing multi-qubit outputs. The problem is that we do not have any such dataset, so we will somehow have to make our own. In practice, each of the four sections of the device in Figure 5b have their own resonant frequency. The method is to then prepare an input signal containing all the desired resonant frequencies and send it into the device. Each section will then return a reflected signal that gets mixed together with the others, meaning all information is stored in a single output signal. With this in mind, we will be able to mimic multi-qubit readouts by taking existing single-qubit readouts carrying different frequencies and combining them together. However we still only have readouts from one of the DQDs with its single resonant frequency responses.

A possible solution is to not use experiment readouts and instead turn to simulations. This is done by composing a transmission line of impedance $Z_0$ in series with an impedance matching RLC circuit consisting of an inductor $L$ in series with a capacitor $C$ and a resistor $R$ placed in parallel. Here $R$ would be the SQD, while $L$ is used to give the RLC circuit an impedance near $Z_0$ when driven at the resonant frequency. We can then simulate the signal reflected off the RLC circuit and use that as a qubit readout.

While the simulation we have only generates single-qubit reflected signals, we do have full control of the resonant frequencies, meaning we can use the previous idea of mixing them together to get a multi-qubit readout. We have created three-qubit signals with the resonant frequencies used in the device from Figure 5b, which are 130 MHz, 176 MHz, and 158 MHz for the top left, bottom left, and bottom right qubits respectively. The signals run for one microsecond using a sampling rate of 1.8 GHz with an adequate SNR to properly showcase PCA on multiple qubits.

The dataset variable space is the same as before, but now PCA should project onto a three-dimensional space to attain three features. We therefore get three principal components $w_1, w_2, w_3$ used to classify each qubit in the signal individually, i.e. $w_1$ would create a splitting for the first qubit, but say nothing about the other qubits. Because of this, we see a splitting around zero along each axis in the PCA subspace, creating a cluster in every octant as seen in Figure 11. The individual classification accuracies for the 130 MHz, 176 MHz, and 158 MHz qubits are $99.9\%, 97.6\%, 90.5\%$ respectively, while the overall accuracy where all three qubits are determined correctly at the same time is 88.3%.

From these results it is clear PCA can be used for classifying multiple qubits from the same signal. We could also have included the last qubit from the device with a resonant frequency of 139 MHz, but that would have made it difficult to visualize with a plot. The major downside of using PCA is that it is unsupervised, so it will not be obvious what principal component corresponds to what qubit or which sign the $|0\rangle$ and $|1\rangle$ qubit states should get assigned to for each qubit. The first problem can be solved by finding the frequency of the principal components through a Fourier transform as we have done before. The second problem would be difficult to get around without having the states already labeled as we would otherwise need to know the amplitude and phase returned by the states beforehand to make a proper judgement. Luckily we will only need to use labels once to figure this out, meaning the results of all subsequent experiments can easily be determined.

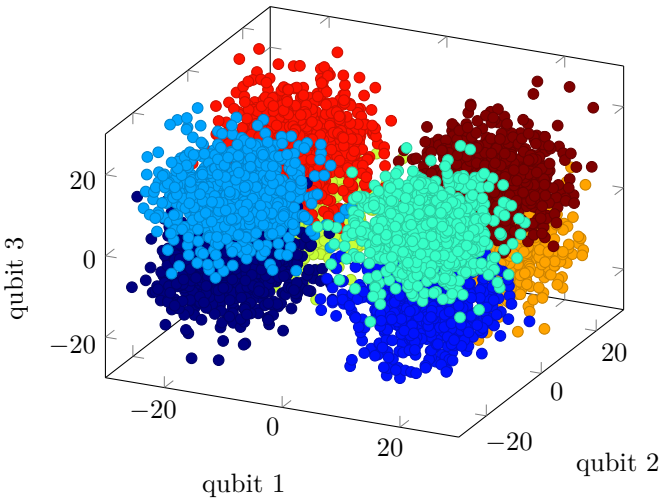



Figure 11: The PCA transformation of simulated signals carrying information from three different qubits. The labels qubit 1, qubit 2, and qubit 3 refer to the qubits with 130 MHz 176 MHz, and 158 MHz resonant frequencies respectively. The signals are clustered into separate octants, where each octant represents a three-qubit state, e.g. $|011\rangle$. The transformed signals are colored according to the labels they were given during simulation.

# 4 Dynamical Quantum Phase Transitions

Phase transitions are usually studied using some controllable parameter like temperature while the system is at equilibrium. We will instead look at phase transitions that occur from time elapsing. This means the parameter is not under our control and the system is not in equilibrium. To be specific, we will study so-called Dynamical Quantum Phase Transitions (DQPTs) under a sudden quench, which is the topic of [4, 12] and this introduction will be largely based off of these. Our system will start in the ground state $|\psi_0\rangle$ of $H_0$, and then we will instantly change the Hamiltonian to $H$ to perform a quench. As a consequence, the system will become dynamic, assuming $|\psi_0\rangle$ is not an eigenstate of $H$. It is possible that the ground state is degenerate, but we will assume this is not the case to keep things simple. The system evolves in accordance with the time evolution operator $U$, meaning the quenched state can be written as $|\psi_0(t)\rangle = U|\psi_0\rangle = e^{-iHt}|\psi_0\rangle$.

To understand what a DQPT is, we first need to get some definitions out of the way. The first one is the Loschmidt amplitude

$$\mathcal{G}(t) = \langle\psi_0|\psi_0(t)\rangle = \langle\psi_0|U|\psi_0\rangle = \langle\psi_0|e^{-iHt}|\psi_0\rangle\,,$$

which tells us how much $|\psi_0(t)\rangle$ overlaps with $|\psi_0\rangle$ at any point in time. Because of the similarity to the partition function $Z = \operatorname{tr} e^{-\beta H}$, we can try and define something similar to the free energy density $f$ by taking inspiration from its definition $Z = e^{-\beta N f}$, where $N$ is the number of degrees of freedom. This results in the relation $\mathcal{G}(t) = e^{-Ng(t)}$, giving rise to the function $g(t)$. As the free energy (density) can be used to find phase transitions by looking for nonanalyticies, we will in the same spirit define a DQPT as a point in time $t^*$ where $g(t)$ becomes nonanalytic. A DQPT is not necessarily a sign of a change in a physical observable like the free energy, but instead an indication of a sudden drastic change in the time evolution operator $U$. However, the reason for this change cannot pinned down by simply looking at $g(t)$ and will therefore require a deeper analysis to get an answer. A natural modification of the Loschmidt amplitude would be the probability $\mathcal{L}(t) = |\mathcal{G}(t)|^2$ named the Loschmidt echo with the associated rate function $\lambda(t)$ from $\mathcal{L}(t) = e^{-N\lambda(t)}$. The rate function can also be expressed by $\lambda(t) = 2\operatorname{Re} g(t)$ and therefore be used to find DQPTs too.

## 4.1 Transverse Field Ising Model

Currently all we have is the definition of a DQPT, but we will also need some data to implement machine learning. For this we choose to look the Transverse Field Ising Model (TFIM)

$$H = -J\sum_{l=1}^{L}\sigma_l^x\sigma_{l+1}^x - \Gamma\sum_{l=1}^{L}\sigma_l^z, \tag{15}$$

as it is possible to find an analytic solution to its rate function $\lambda(t)$. The derivation of the rate function is found in Apx. B, but we give a quick overview of the steps and results. First a Jordan–Wigner transformation is performed to fermionize the Hamiltonian by writing the Pauli matrices in terms of fermionic operators $c_l^\dagger$ and $c_l$. The problem is then brought into momentum space with a Fourier transform followed by a Bogoliubov rotation that gives us the fermionic operators $\eta_k^\dagger$ and $\eta_k$ which diagonalize the Hamiltonian

$$\sum_k \omega_k\eta_k^\dagger\eta_k + \text{const.} \quad \text{with} \quad \omega_k = 2\sqrt{J^2 - 2J\Gamma\cos(k) + \Gamma^2}. \tag{16}$$

Assuming we quench from $(J,\Gamma)$ to $(J',\Gamma')$, the rate function in the thermodynamic limit is

$$\lambda(t) = -\frac{1}{2\pi}\int_0^\pi \log\bigl(1 - \sin^2(2\phi_k)\sin^2(\omega_k' t)\bigr)\,\mathrm{d}k \quad \text{where} \quad \sin(2\phi_k) = \frac{4(J\Gamma' - J'\Gamma)\sin(k)}{\omega_k\omega_k'}, \tag{17}$$

with the additional result that DQPTs will occur if and only if $|\Gamma/J| < 1 < |\Gamma'/J'|$ or visa versa and they do so at times

$$t_n = \frac{\pi(n+1/2)}{\omega_{k^*}'} \quad \text{where} \quad \omega_{k^*}' = 2\sqrt{(J'+\Gamma')(J'-\Gamma')\frac{(\Gamma/J) - (\Gamma'/J')}{(\Gamma/J) + (\Gamma'/J')}}. \tag{18}$$

In the subsequent sections we will train neural networks on the rate function and that requires us to compute it. The integral in Eq. 17 is not easily solvable, so we instead discretize it by letting the system

size $L$ stay finite. This lets us compute an approximation of the rate function using any parameters we wish to use. In our case we choose a system size of $L = 512$ which shows very clear DQPTs. For most of the neural networks, we will only sample $N = 512$ points from the rate function where each sample is spaced $\Delta T = 0.008$ apart for a total time of $T = \Delta T N = 4.096$. As for the parameters of the Hamiltonian, we set $J = J' = 1$ as only their ratios with $\Gamma$ and $\Gamma'$ matter for the general shape of the rate function. To get enough variety, we let the system start within the uniform distributions $\Gamma \sim U(0, 1)$ whereafter we quench to either $\Gamma' \sim U(0, 1)$ or $\Gamma' \sim U(1 + \pi/T, 4)$ to get traces with both no DQPTs and those with DQPTs. The extra term $\pi/T$ is used to ensure at least one DQPT will appear within a time $T$ to avoid confusing the models under training. It is also necessary to normalize all of the traces for our purposes. This is because if the rate function has a DQPT, then its maximum value is generally greater than those without DQPTs. This would go against the idea of trying to get a neural network to learn the various characteristics of the rate function when a single sample in the trace could be used produce a classification.

## 4.2 Classification of Loschmidt Echos

While PCA can work well for signals when classifying qubits, it does not actually compare the samples in a signal in any meaningful way as it is simply an affine transformation. A multi-layered FNN would be a good substitute to PCA in that regard, but for inputs that are more complex, it could severely hurt the interpretability of the neural network. We therefore choose to work with an RNN instead as it only has a few variables and has an output at every timestep, creating its own curve as it goes along the input trace. This makes it possible to visually see what the RNN is doing, allowing for easier interpretation. We will be using a two-layered RNN, where the first layer takes the one-dimensional input value at each timestep and has a five-dimensional hidden state. The second layer takes the hidden state from the previous layer and returns a two-dimensional hidden state. We then apply the softmax function on the last hidden state of the RNN to get a probability distribution. These two probabilities will be the output of the RNN and signify whether a trace either did or did not contain a DQPT.

To train the model we use the cross-entropy loss function which is designed for training these kind of probability outputs. The function expects the output to not have been softmaxed yet, so that part of the model will be removed during training. From testing, this loss function has a hard time getting anywhere without prior training, so to kickstart it we add a term to encourage the model to refrain from giving a fifty-fifty output. This is done by computing the standard deviation across the output of the current training batch holding $N$ entries (with softmax applied):

$$s_i = \sqrt{\frac{1}{N-1}\sum_{n=1}^{N}\left(\mathrm{softmax}(h_T)_{ni} - \overline{\mathrm{softmax}(h_T)}_{ni}\right)^2}. \tag{19}$$

In our case $s = (s_1, s_2)$, but both entries will be around the same value because the model output essentially gets mirrored around 0.5 by the softmax function. Therefore we choose to only consider $s_1$. We want to maximize $s_1$ as the max standard deviation of 0.5 would mean an even distribution of zero and one probabilities for the first entry of the output. To translate this idea into the loss function we subtract $160s_1$ from the current loss function, but only when $s_1 < 0.45$ as the factor 160 dominates over the cross-entropy loss function. That way the model quickly moves away from a fifty-fifty output at the start before training with the cross-entropy loss function, which is what really makes the model learn to classify DQPTs. To apply the model we trained it on 5000 different traces of both types and it yielded a 100% classification accuracy on the validation traces.

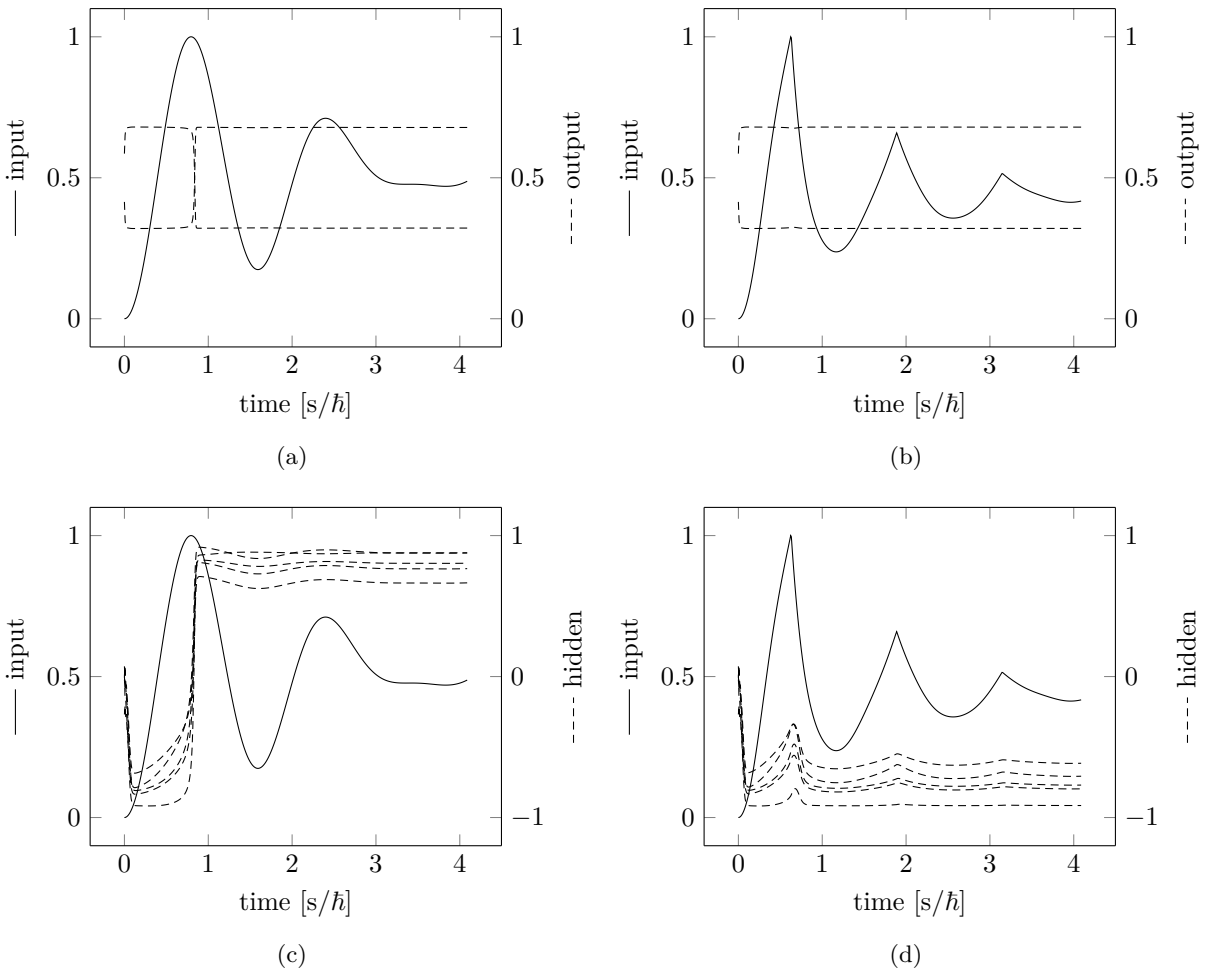



Figure 12: Two rate functions, one lacking DQPTs with $(\Gamma_0, \Gamma) = (0.153, 0.274)$ and one displaying three DQPTs with $(\Gamma_0, \Gamma) = (0.533, 1.935)$, shown left and right respectively. **(a,b)** The outputs of the RNN displayed together with the input rate function. The outputs switch places if the RNN encounters a peak without a kink. **(c,d)** The hidden states of the first layer also displaying a change in overall position when a peak without a kink is reached and whereafter they stay there until the end.

Two different looking traces can be seen in Figure 12a and 12b with both the input trace and the two softmax outputs plotted as a function of time. The outputs never achieve values anywhere close zero or one as one would expect for a model that is confident in its answer. However, this is not because of any uncertainty, but rather a result of the RNN using tanh as its non-linear function which only returns values between $-1$ and 1, resulting in the softmax function never reaching its extremes. The model outputs always start out by going towards the bottom and the top. After that both curves stay flat until the input function peaks and the model decides if there is a DQPT or not. Hereafter the outputs stay constant again until the last timestep where the output component with the highest probability is chosen as the model classification. This is all pretty vague and it does not help us understand much about how the model is able to classify the traces so well. Luckily, the first hidden layer of the RNN can be of help. The five hidden curves are plotted in Figure 12c and 12d. Here it is still hard to find anything noteworthy that helps us figuring out how the model works. However there is one feature that jumps out, namely the small increase in value of the hidden curves around the kinks in Figure 12d. For the time being it is not clear why it is important, so we instead try to pass some simpler inputs to the model that will make it possible to understand how the model classifies the rate functions.

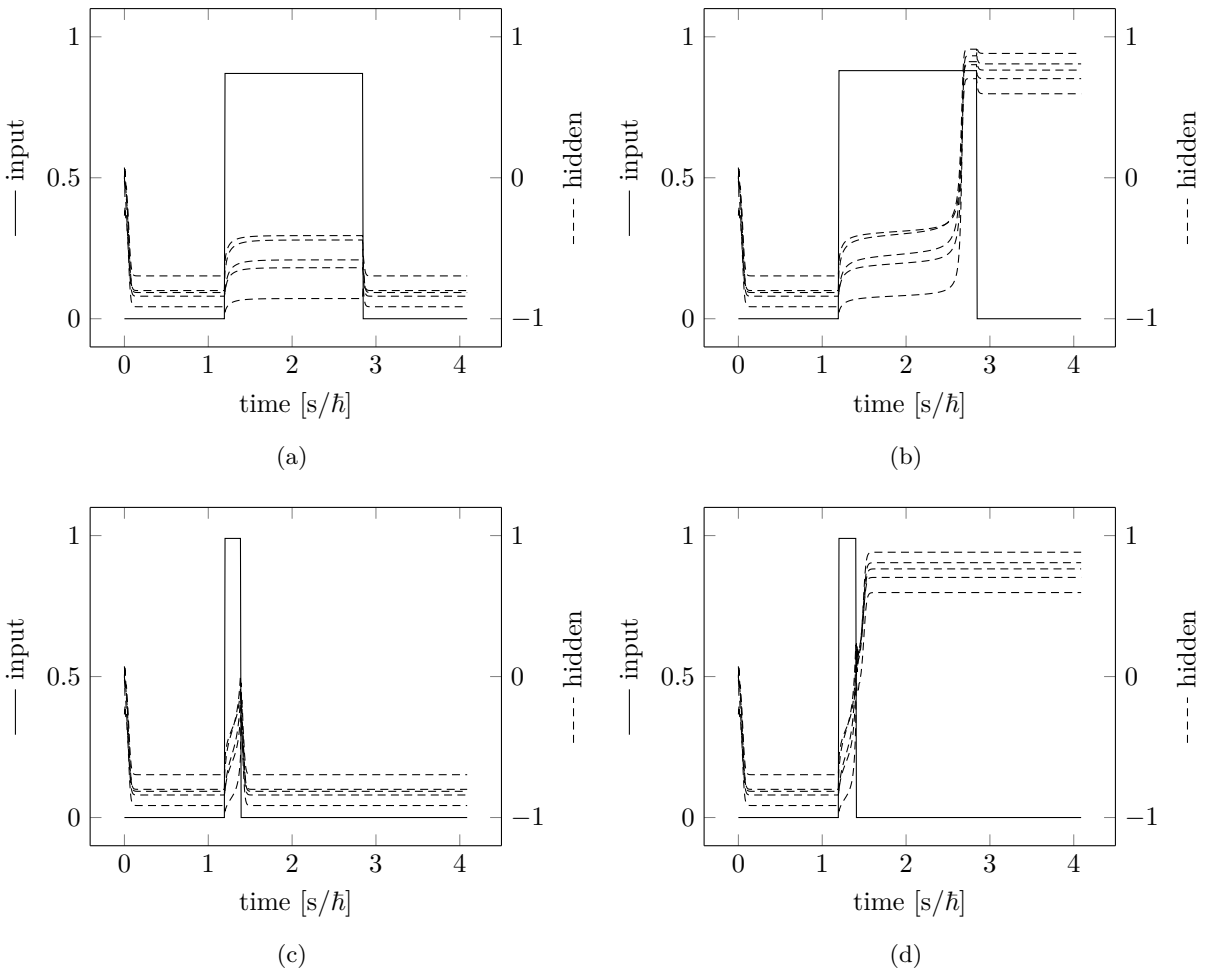



Figure 13: Various square pulse inputs to probe the behaviour of the RNN and its hidden states. **(a,b)** The inputs use a fixed width of 1.65 s/$\hbar$ and heights of 0.87 and 0.88 in the order they are shown. The hidden states travels through the lowest pulse, but stays near the bottom throughout the test. On the other hand, using the higher pulse, the hidden states travel to the top and stay these even after the pulse has ended. **(c,d)** For the second pair a height of 0.99 is used, while the lengths are 0.19 s/$\hbar$ and 0.21 s/$\hbar$. A similar scenario takes place where the hidden states stay at the bottom for one, while they go to the top for the other.

There are two types of handmade curves that we give the model. The first ones are shown in Figure 13a and 13b and give an insight into how the first hidden layer depends on the immediate value of the input. The inputs consists of a single square pulse of varying height, but fixed duration. When the square pulse occurs, the hidden curves begin to rise together until they either flatten out or suddenly jump up. After the square pulse ends, the hidden curves will either go back down to where they initially were or stay fixed at the top if they decided to perform the jump. It is the placement of the hidden curves that controls whether the model thinks there is a DQPT or not. If their end value is low, then the curve has DQPTs, while a high value means there were no DQPTs at any point in the input. The sudden jump in Figure 13b occurs after changing the square pulse height from 0.87 (Figure 13a) to 0.88, which shows there is some kind of build-in threshold value. This means the model only looks at the peaks of the input, as no points that are lower than 0.88 would be able to change the output of the model.

We now turn to look at Figure 13c and 13d, where instead of raising the height of the square pulse, it is instead the width that gets changed. In this case the height is fixed to 0.99 and it is a small change in the width that allows the hidden curves to suddenly jump. The cause of this can be viewed as the hidden layer needing to "charge" up to a specific value under the influence of the input. If it is allowed to charge for long enough, it will be able to jump and stay there, but if the input dips prematurely, it will sink down again to its previous height.

We can now put the ideas behind the two sets of figures together and deduce how the model functions. The two main points are that the hidden layer only "charges" whenever the input is high enough, but

for the output to change the input also needs to stay high for long enough. The characteristics of DQPT kinks are that they rise very fast, top, and then fall just as fast again. From our current understanding, a DQPT will therefore not change the model output because the hidden layer does not have enough time to reach a sufficiently high value that allows for it to jump. On the other hand, rate functions without DQPTs will not have these kinks, and by extension will have peaks that top for a longer time. This will cause the hidden layer to jump and stay up there for the rest of the input, signaling to the model that there will not be any DQPTs.

The mechanism behind the jump can be understood using attractive fixed points. These are a type of point that an iterated function sequence will converge towards if the first point in the sequence is in a neighboring region of it. In our case, the first layer of the RNN has two 5-dimensional attractive fixed points, namely

$$p_- = \begin{pmatrix} -0.9149 \\ -0.8401 \\ -0.8140 \\ -0.8000 \\ -0.6961 \end{pmatrix} \quad \text{and} \quad p_+ = \begin{pmatrix} 0.8820 \\ 0.5960 \\ 0.7638 \\ 0.8079 \\ 0.7033 \end{pmatrix}. \tag{20}$$

These were estimated using inputs similar to Figure 13 where input rests at zero at the end of a pulse. Other resting values close to zero would simply skew the attractive fixed points a bit. From this, if the first layer has a value close to $p_-$ at the end, then the trace contains DQPTs, while a final value close to $p_+$ would mean it is devoid of DQPTs. In Figure 13 we only looked at positive square pulses that sent the hidden layer from $p_-$ to $p_+$, but with a negative pulse it is also possible to go from $p_+$ to $p_-$.

Another more obvious way to detect DQPTs would be to calculate the second derivative at each point of the input and then see if it goes below some threshold. This should work well as inputs with a DQPT have very apparent kinks compared to those that lack DQPTs. It is even possible to make this into an RNN by creating one by hand. Using that the second derivative can be computed discretely with $D_t^2 = x_{t+1} - 2x_t + x_{t-1}$, we can write out the RNN components

$$W_i = \begin{pmatrix} 1 \\ 0 \\ -(1) \\ 0 \end{pmatrix}, \quad W_h = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ -(-2) & -(1) & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} M \\ 0 \\ -M - T \\ 0 \end{pmatrix}. \tag{21}$$

Here $T$ is the threshold (taken to be positive), $M = -\min\{x_t\}$, and we choose ReLu as the non-linearly of the model as it works well with the thresholding method. To start, assume we are already in the middle of the input at $t+1$. Ignoring the ReLu function at the moment, the calculation for the hidden state at $t+1$ will be

$$W_i x_{t+1} + W_h h_t + b = \begin{pmatrix} 1 \\ 0 \\ -(1) \\ 0 \end{pmatrix} x_{t+1} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ -(-2) & -(1) & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_t + M \\ x_{t-1} + M \\ V_t \\ O_t \end{pmatrix} + \begin{pmatrix} M \\ 0 \\ -M - T \\ 0 \end{pmatrix} \tag{22a}$$

$$= \begin{pmatrix} x_{t+1} + M \\ x_t + M \\ -(x_{t+1} - 2x_t + x_{t-1}) - T \\ V_t + O_t \end{pmatrix} \tag{22b}$$

$$= \begin{pmatrix} x_{t+1} + M \\ x_t + M \\ -D_t^2 - T \\ V_t + O_t \end{pmatrix}. \tag{22c}$$

It is seen that $h_t$ is set to something very specific because we are already in the middle of the trace. To get a proper understanding of it, we go through its terms one by one. First is $x_t + M$, whose entry is used to store the input from time $t$ directly in the hidden state. There is the additional term $M$ in there, which is used to ensure the value is non-negative. This prevents the value from being changed after ReLu is applied and therefore ensures the calculation of the second derivative is correct. The second term is the same as the first term, except it is used to store $x_{t-1}$. Next is the third entry $V_t$. Its value is simply $\text{ReLu}(-D_{t-1}^2 - T)$, but it is here the chosen non-linearly ReLu is crucial. If the second derivative is not low enough then $V_t$ is simply zero, so it is only when $D_{t-1}^2 < -T$ that $V_t$ becomes positive. This way $V_t$

actually stores whether the input has a kink that reaches our threshold $T$. The fourth entry $O_t$ is used to remember if $V_t$ was positive at any point by adding $V_t$ to itself at each timestep. After the RNN has processed the whole input we then read out $O_t$ at the last timestep and check if it is positive. In the case it is, we know there was a kink sharp enough to reach our threshold.

Right now we are only looking at a point in the middle of the input, but to actually run the RNN we also need to ensure it starts out properly. This is done by initializing the hidden state at

$$h_0 = \begin{pmatrix} M \\ M \\ 0 \\ 0 \end{pmatrix}, \tag{23}$$

as both $D_0^2$ and $D_1^2$ will then be non-negative, ensuring the threshold will never be reached right at the start of the input. By using a threshold of $T = 0.015$ it is possible to get an accuracy as high as 99.8% on the same validation traces that were used to test the trained RNN. Just from this, it seems thresholding the second derivative is slightly worse at classifying the generated traces, but this is a bit misleading. Instead it is a result of the traces being generated using single precision floats, which causes the generator program to once in a while produce very jarring outputs with multitudes of kinks. Redoing the traces with double precision floats smooths out the erroneous traces and raises the accuracy up to 100%. Even though this was the fault of the generator and not the model, it still shows this is not the most robust method of classifying as any sudden kink in the trace could lead to a wrong classification. On the other hand, the trained RNN does not suffer from this problem as it instead uses the general shape of the trace for classification. That way there will not be any misclassification, even if it has some discrepancies compared to the genuine trace.

### 4.3 Long Short-Term Memory

In the previous section we only used one of the most basic RNNs, but this raises the question of what more complex RNNs can accomplish. We will therefore be implementing an LSTM in this section to use its increased memory capabilities. To make proper use of the increased complexity of an LSTM, we will refrain from simply classifying traces and instead train an LSTM to point out where each DQPT happens. The actual output of the model should produce a spike whenever a DQPT happens, but stay at zero otherwise. To make it do this we produce a target output that the model should try to match for each trace. Explicitly the whole target output is zero, except at the exact points where a DQPT occurs, where it takes the value 0.4 instead. This spike value was chosen to be 0.4 because the LSTMs had problems with outputting higher values than this when trained. The loss function also needs to be very different as it is no longer only the last timestep that is used, but instead the whole output of the RNN. We will use a loss function that was found through trail and error:

$$\frac{1}{T}\sum_{t=1}^{T} \exp(7T_t + 10|O_t - T_t|). \tag{24}$$

Here $O$ is the model output and $T$ is the target output with the subscripts denoting the time. The general idea behind this loss function is to harshly punish the model for being very far away from the target output at any point. Because there are so few spikes in each trace with a DQPT it is also necessary to include the extra $7T_t$ term in the exponential, otherwise the model would more or less just ignore the spikes in favour of producing a flattened output.

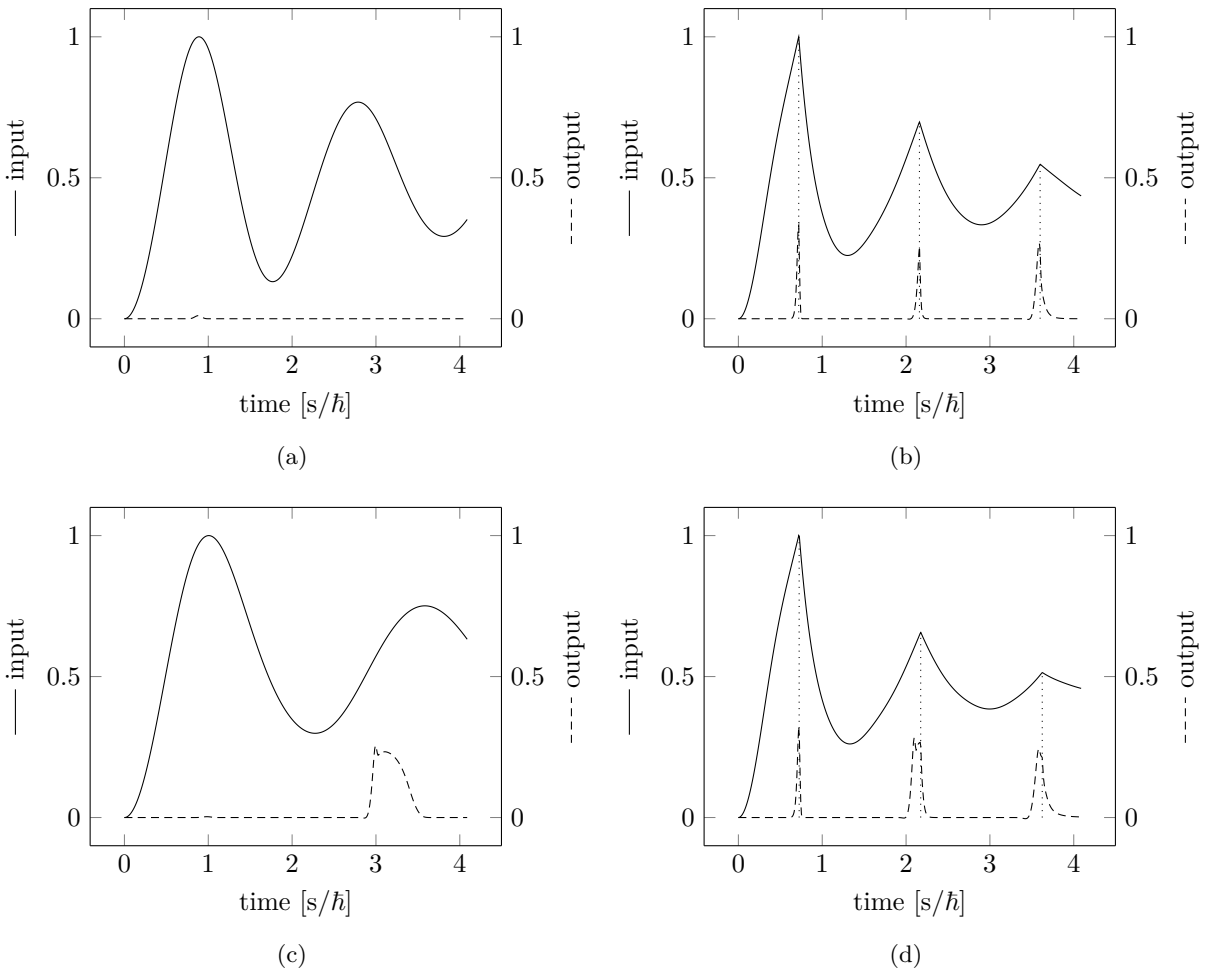



Figure 14: Plots of the LSTM marking DQPTs with spikes in its output. **(a,b)** The LSTM is able to produce sharp spikes, but stays at zero otherwise. **(c,d)** The spikes are wider than wanted and the trace without any DQPTs makes the LSTM produce a small lump in its output where nothing should be happening.

To test how well LSTMs can do, we will employ a two-layered LSTM where the first layer is two-dimensional and the second layer is one-dimensional to match output we want. The results can be seen in Figure 14. The model is able to pick up on all of the kinks in a trace with DQPTs, while those without generally stay flat around zero. Prime examples are seen in Figure 14a and 14b. It does fail slightly in some cases though. In Figure 14a there is a small bump around the first peak of the input, but that can be considered negligible as the output never reaches a sizeable fraction of 0.4. Contrarily, Figure 14c shows a large bump whose peak is comparable to those in Figure 14b. The model can also have problems producing a proper peak at a DQPT, which should in idea be represented by a sharp incline just before it happens and then a sharp decline right after. This does not happen for all DQPTs in Figure 14d where the last two peaks start their incline and peak before the DQPTs happen. Luckily, the peaks do decline sharply right after the DQPTs as expected.

An explanation for the behaviour of the model in Figure 14c and 14d is that it cannot predict the rest of the trace ahead of time. For this reason, it has to produce a sharp incline whenever it thinks a DQPT might occur. From here it stays elevated until it is sure a possible DQPT has passed. In the case there was no DQPT, it slowly falls back down to zero as the model has no proper indicator whether its guess was incorrect or not. It can however spot DQPTs very easily and will immediately descend towards zero when they occur, albeit its initial guess for the start of the peak can be slightly off.

The model can also be used as a classifier despite it not being the intention. It is turned into a classifier by simply checking if the model output breaks a threshold at any point. If that happens the trace is considered to be containing DQPTs as those that actually do should have peaks that go above the threshold. Traces without DQPTs should of course stay near zero and therefore never go reach the threshold, but it was seen in Figure 14c that some traces break this notion. Despite this, it is possible

to work out a threshold of 0.26, which barely yields us the classification accuracy of 100%. Thresholds below 0.26 or above 0.27 do not give us this accuracy.

# 5 Conclusion

We started off in Chapter 2 by introducing the machine learning methods we were going to use. This included principal component analysis that tried to separate a dataset by finding the axes with most variance. By projecting the data onto these axes with an affine transformation we could get a much better view of the data clustering that would otherwise be shrouded by the amount of dimensions used to represent a sample in the dataset. Then we delved into neural networks whose variability allow for the classification of data that cannot be handled through a simple projection. We established the feedforward neural network as a starting point because of its relative simplicity and how it could be trained for use in classification problems. We then expanded upon the FNN with recurrent neural networks, going through both the standard RNN and the more complex long short-term memory variant.

Our first usage of machine learning was in Chapter 3 where we used PCA to classify qubits. Our aim was to try and beat the usual readout time of several microseconds accomplished with homodyne detection. From the experimental data we had at hand, PCA was able to get readout times down to one microsecond while nearly achieving full accuracy in its classification. It was also possible to go below this time by systematically discarding signals we were unsure about. By analysing the principal component we found that the projection we were applying worked much like homodyne detection, but PCA was able to artificially produce a signal that yielded better results. The last part of the chapter showed the capability of PCA for classifying readout signals containing information about multiple qubits. The signals we used for this consisted of multiple simulated single qubit readout signals added together, but there is no guarantee that the qubits would not be coupled to each other in some way. Therefore our use of PCA on multiple qubits was more of a showcase of how it would be done in practice and a full multi-qubit simulation would be needed to properly test PCA.

Then we set our sights towards dynamical quantum phase transitions in Chapter 4 by training RNNs on Loschmidt rate functions. By letting a basic RNN observe enough traces with and without DQPTs, we were able to classify whether or not a DQPT occurred in a trace. We then went through some handmade traces to try and understand what the RNN was doing. It turned out that the network looked for intervals with a high value that stayed long enough, meaning those intervals could not contain a kink that only peaks for an instant. This behaviour could be understood through attractive fixed points that the hidden states would go towards if close enough. We also sculpted an RNN that used the second derivative to find kinks, which had problems with traces that contained some jarring features. After that we tried to make a long short-term memory RNN into a DQPT locator by showing a spike in its output whenever it detects the associated kink. There were a few traces that gave it trouble, for example it produced lump in its output for a trace that did not contain any DQPTs. Getting a LSTM to properly detect DQPTs therefore requires further work.

# A Principal Component Analysis Solution

Let $U$ be a real $l$-dimensional vector space which the $n$ samples $\{x_i\}_{i\in[n]}$ live in. By assuming $\bar{x} = 0$, we can construct the $n \times l$ sample matrix $X$ and define the variance for a variable $v$

$$\mathrm{Var}(v) = v^{\mathrm{T}}\mathrm{Cov}(X)v = (O^{\mathrm{T}}v)^{\mathrm{T}}\Lambda(O^{\mathrm{T}}v), \tag{25}$$

where $\mathrm{Cov}(X) = \frac{1}{n-1}X^{\mathrm{T}}X$ is the sample covariance matrix of $X$. Here we have diagonalized $\mathrm{Cov}(X)$ with the orthogonal matrix $O$ where $\lambda_i = \Lambda_{ii}$ are the eigenvalues and the columns of $O$ are the eigenvectors $\{u_i \mid i = 1, \ldots, l\}$. We will assume the eigenvalues and eigenvectors are ordered by highest eigenvalue such that $\lambda_i \geq \lambda_j$ if $i \leq j$.

Let $V$ be a $p$-dimensional subspace of $U$ with eigenvectors $\{v_i \mid i = 1, \ldots, p\}$ with the restriction $p \leq l$. We define the variance of this space by

$$\mathrm{Var}(V) = \sum_{i=1}^{p} \mathrm{Var}(v_i), \tag{26}$$

which is invariant under basis change. The goal is to show $\mathrm{Var}(V)$ is maximized by choosing $V = \mathrm{span}\{u_i \mid i = 1, \ldots, p\}$, proving the PCA algorithm maximises the explained variance.

With $P_V$ being the operator for projecting vectors onto $V$, we can define $w_j = \|P_V u_j\|^2$ that possess the inequality $w_j \leq 1$ since $\|P_V u_j\|^2 \leq \|u_j\|^2$. It can also be written in terms of the basis for $V$ through some manipulations

$$w_j = \|P_V u_j\|^2 = \left\| \sum_{i=1}^{p} \langle u_j, v_i \rangle v_i \right\|^2 = \sum_{i=1}^{p} \langle u_j, v_i \rangle^2 = \sum_{i=1}^{p} (u_j \cdot v_i)^2 = \sum_{i=1}^{p} \left( \sum_{k=1}^{l} (O^{\mathrm{T}})_{jk} (v_i)_k \right)^2 = \sum_{i=1}^{p} (O^{\mathrm{T}}v_i)_j^2, \tag{27}$$

which lets us express the subspace variance in a different manner

$$\mathrm{Var}(V) = \sum_{i=1}^{p} \mathrm{Var}(v_i) = (O^{\mathrm{T}}v_i)^{\mathrm{T}}\Lambda(O^{\mathrm{T}}v_i) = \sum_{i=1}^{p}\sum_{j=1}^{l} \lambda_j (O^{\mathrm{T}}v_i)_j^2 = \sum_{j=1}^{l} w_j\lambda_j = \sum_{j=1}^{p} w_j\lambda_j + \sum_{p+1=1}^{l} w_j\lambda_j. \tag{28}$$

From here we further define $W = \sum_{j=1}^{p} w_j$ and $W^{\perp} = \sum_{j=p+1}^{l} w_j$ with their total

$$W + W^{\perp} = \sum_{j=1}^{l} w_j = \sum_{i=1}^{p} \left( \sum_{j=1}^{l} (O^{\mathrm{T}}v_i)_j^2 \right) = \sum_{i=1}^{p} \left\| O^{\mathrm{T}}v_i \right\|^2 = p, \tag{29}$$

where we used that $O$ is orthogonal. By using $w_j \leq 1$ to ensure $1 - w_j \geq 0$ and that $\lambda_i \geq \lambda_j$ when $1 \leq i \leq p < j \leq l$, we arrive at the important inequality

$$\sum_{p+1=1}^{l} w_j\lambda_j = \left( \frac{1}{p - W} \sum_{k=1}^{p} (1 - w_k) \right) \sum_{j=p+1}^{l} w_j\lambda_j \tag{30}$$

$$\leq \frac{1}{W^{\perp}} \sum_{k=1}^{p} (1 - w_k) \sum_{j=p+1}^{l} w_j\lambda_k = \left( \frac{1}{W^{\perp}} \sum_{j=p+1}^{l} w_j \right) \sum_{k=1}^{p} (1 - w_k)\lambda_k = \sum_{k=1}^{p} (1 - w_k)\lambda_k. \tag{31}$$

This leads us directly to our goal

$$\mathrm{Var}(V) \leq \sum_{j=1}^{p} w_j\lambda_j + \sum_{j=1}^{p} (1 - w_j)\lambda_j = \sum_{j=1}^{p} 1 \cdot \lambda_j = \sum_{j=1}^{p} \mathrm{Var}(u_j) \tag{32}$$

that shows us $V = \mathrm{span}\{u_i \mid i = 1, \ldots, p\}$ has the maximum variance for a $p$-dimensional subspace of $U$.

# B Transverse-Field Ising Model Quench Solution

The goal of this section is to derive the rate function $\lambda(t)$ of the Loschmidt echo for the Transverse-Field Ising model. This is done by first diagonalizing the model and then quenching it to get the Loschmidt echo $\mathcal{L}(t)$. The Hamiltonian for the Transverse-Field Ising model is

$$H = -J\sum_{l=1}^{L}\sigma_l^x\sigma_{l+1}^x - \Gamma\sum_{l=1}^{L}\sigma_l^z. \tag{33}$$

## B.1 Jordan-Wigner Transformation

We start the diagonalization by using the Jordan-Wigner transformation which lets us replace the Pauli operators $\sigma_l^z, \sigma_l^-, \sigma_l^+$ with fermion operators $c_l^\dagger, c_l$:

$$\sigma_l^z = 1 - 2c_l^\dagger c_l, \quad \sigma_l^- = \exp\left(i\pi\sum_{j<l}c_j^\dagger c_j\right)c_l^\dagger, \quad \sigma_l^+ = \exp\left(i\pi\sum_{j<l}c_j^\dagger c_j\right)c_l. \tag{34}$$

Before we apply this transformation to Eq. 33, we first note

$$e^{i\pi c_l^\dagger c_l} = \sum_{n=0}^{\infty}\frac{(i\pi c_l^\dagger c_l)^n}{n!} = 1 + \sum_{n=1}^{\infty}\frac{(i\pi)^n}{n!}c_l^\dagger c_l = 1 + (e^{i\pi}-1)c_l^\dagger c_l = 1 - 2c_l^\dagger c_l = -1 + 2c_l c_l^\dagger, \tag{35}$$

from which we get the useful relations $c_l^\dagger = c_l^\dagger e^{i\pi c_l^\dagger c_l} = -e^{i\pi c_l^\dagger c_l}c_l^\dagger$ and $c_l = -c_l e^{i\pi c_l^\dagger c_l} = e^{i\pi c_l^\dagger c_l}c_l$. Replacing $\sigma_l^z$ is straight forward for any $l$, so we immediately shift our focus to $\sigma_l^x\sigma_{l+1}^x$ which requires some extra care. This is because we want to use a Fourier transformation later and that necessitates approximating the model using boundary conditions. Before we go into exactly what boundary condition to use, we first do the substitutions for $l < L$ as those can be done without additional considerations

$$\begin{aligned}
\sigma_l^x\sigma_{l+1}^x &= (\sigma_l^- + \sigma_l^+)(\sigma_{l+1}^- + \sigma_{l+1}^+) && \text{(36a)}\\
&= \left(e^{i\pi\sum_{j<l}c_j^\dagger c_j}c_l^\dagger + e^{i\pi\sum_{j<l}c_j^\dagger c_j}c_l\right)\left(e^{i\pi\sum_{j<l+1}c_j^\dagger c_j}c_{l+1}^\dagger + e^{i\pi\sum_{j<l+1}c_j^\dagger c_j}c_{l+1}\right) && \text{(36b)}\\
&= c_l^\dagger e^{i\pi c_l^\dagger c_l}c_{l+1}^\dagger + c_l^\dagger e^{i\pi c_l^\dagger c_l}c_{l+1} + c_l e^{i\pi c_l^\dagger c_l}c_{l+1}^\dagger + c_l e^{i\pi c_l^\dagger c_l}c_{l+1} && \text{(36c)}\\
&= c_l^\dagger c_{l+1}^\dagger + c_l^\dagger c_{l+1} - c_l c_{l+1}^\dagger - c_l c_{l+1} && \text{(36d)}\\
&= (c_l^\dagger - c_l)(c_{l+1}^\dagger + c_{l+1}). && \text{(36e)}
\end{aligned}$$

Now, it is for $l = L$ we need to take boundary conditions into account which depends on the fermionic parity $P = e^{i\pi\sum_{j=1}^{L}N_j}$ where $N_j = c_j^\dagger c_j$ is the number operator at site $j$. For even fermionic parity we will choose an anti-periodic boundary condition $c_{L+1} = -c_1$, while for uneven parity we opt for $c_{L+1} = c_1$ which is periodic. If the fermionic parity is $p$ we can express these boundary conditions compactly as $c_{L+1} = -pc_1$. This is valid to do because all the fermion operators comes in pairs, which preserves the fermionic parity. The reason for a choosing boundary condition that depends on the fermionic parity becomes clear when doing the substitution for $l = L$

$$\begin{aligned}
\sigma_L^x\sigma_{L+1}^x &= (\sigma_L^- + \sigma_L^+)(\sigma_{L+1}^- + \sigma_{L+1}^+) && \text{(37a)}\\
&= \left(e^{i\pi\sum_{j=1}^{L-1}c_j^\dagger c_j}c_l^\dagger + e^{i\pi\sum_{j=1}^{L-1}c_j^\dagger c_j}c_l\right)\left((-pc_1)^\dagger + (-pc_1)\right) && \text{(37b)}\\
&= -p\left(-e^{i\pi\sum_{j=1}^{L}c_j^\dagger c_j}c_l^\dagger + e^{i\pi\sum_{j=1}^{L}c_j^\dagger c_j}c_l\right)(c_1^\dagger + c_1) && \text{(37c)}\\
&= -pe^{i\pi\sum_{j=1}^{L}c_j^\dagger c_j}(-c_l^\dagger + c_l)(c_1^\dagger + c_1) && \text{(37d)}\\
&= -pp(-c_l^\dagger + c_l)(c_1^\dagger + c_1) && \text{(37e)}\\
&= (c_l^\dagger - c_l)(c_1^\dagger + c_1). && \text{(37f)}
\end{aligned}$$

We now find the Hamiltonian has become

$$H = -J\sum_{l=1}^{L}(c_l^\dagger - c_l)(c_{l+1}^\dagger + c_{l+1}) - \Gamma\sum_{l=1}^{L}(1 - 2c_l^\dagger c_l) \tag{38}$$

## B.2 Fourier Transform

It is clear the next step is the Fourier transform $c_l = L^{-1/2}\sum_k e^{ilk}c_k$, but before this it is important to consider which set of momenta $k$ we are working with. This set depends on the boundary condition we have and it turns out using the anti-periodic boundary condition makes the intermediate calculations and result simpler [13, 14]. We therefore choose to only focus on the system with an even fermionic parity as that is enough for our purposes. The reason this choice is the easiest to work with ends up being because its set of momenta $\{\pm(2n-1)\pi/L \mid n = 1, \dots, L/2\}$ all come in negative-positive pairs, contrary to uneven fermionic parity that has $k = 0, \pi$ as outliers. The set itself is derived by looking at the Fourier transform and noting $c_{L+1} = -c_1$ implies $e^{iLk} = -1$.

To make the Fourier transform easier to manage we split the Hamiltonian into two parts by assigning $H^J = \sum_{l=1}^{L}(c_l^\dagger - c_l)(c_{l+1}^\dagger + c_{l+1})$ and $H^\Gamma = \sum_{l=1}^{L}(1 - 2c_l^\dagger c_l)$. From here we begin transforming $H^J$

$$H^J = \sum_{k_1,k_2}\frac{1}{L}\sum_{l=1}^{L}\left(e^{-ilk_1}c_{k_1}^\dagger e^{-i(l+1)k_2}c_{k_2}^\dagger + e^{-ilk_1}c_{k_1}^\dagger e^{i(l+1)k_2}c_{k_2} + \text{H.c}\right) \tag{39a}$$

$$= \sum_{k_1,k_2}\frac{1}{L}\sum_{l=1}^{L}\left(e^{-il(k_1+k_2)}e^{-ik_2}c_{k_1}^\dagger c_{k_2}^\dagger + e^{-il(k_1-k_2)}e^{ik_2}c_{k_1}^\dagger c_{k_2} + \text{H.c}\right) \tag{39b}$$

$$= \sum_k\left(e^{ik}c_k^\dagger c_{-k}^\dagger + e^{ik}c_k^\dagger c_k + \text{H.c}\right) \tag{39c}$$

$$= \sum_k H_k^J \tag{39d}$$

$$= \frac{1}{2}\sum_k(H_k^J + H_{-k}^J) \tag{39e}$$

$$= \frac{1}{2}\sum_k\Big((e^{ik}c_k^\dagger c_{-k}^\dagger + e^{-ik}c_{-k}^\dagger c_k^\dagger) + (e^{-ik}c_{-k}c_k + e^{ik}c_k c_{-k}) \tag{39f}$$

$$+ (e^{ik}c_k^\dagger c_k + e^{-ik}c_k^\dagger c_k) + (e^{-ik}c_{-k}^\dagger c_{-k} + e^{ik}c_{-k}^\dagger c_{-k})\Big) \tag{39g}$$

$$= \sum_k\Big((i\sin(k)c_k^\dagger c_{-k}^\dagger) + (-i\sin(k)c_k c_{-k}) + (\cos(k)c_k^\dagger c_k) + (1 - \cos(k)c_{-k}c_{-k}^\dagger)\Big) \tag{39h}$$

$$= L + \sum_k \begin{pmatrix}c_k^\dagger & c_{-k}\end{pmatrix}\begin{pmatrix}-\cos(k) & i\sin(k)\\ -i\sin(k) & \cos(k)\end{pmatrix}\begin{pmatrix}c_k\\ c_{-k}^\dagger\end{pmatrix} \tag{39i}$$

We then apply the same techniques to $H^\Gamma$

$$H^\Gamma = L - 2\sum_k c_k^\dagger c_k = L - \sum_k\left(c_k^\dagger c_k + c_{-k}^\dagger c_{-k}\right) = L + \sum_k \begin{pmatrix}c_k^\dagger & c_{-k}\end{pmatrix}\begin{pmatrix}-1 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}c_k\\ c_{-k}^\dagger\end{pmatrix}, \tag{40}$$

and get the fully Fourier transformed Hamiltonian

$$H = -JH^J - \Gamma H^\Gamma \tag{41a}$$

$$= \sum_k \underbrace{\begin{pmatrix}c_k^\dagger & c_{-k}\end{pmatrix}}_{\Psi_k^\dagger}\begin{pmatrix}+(\Gamma - J\cos(k)) & -iJ\sin(k)\\ +iJ\sin(k) & -(\Gamma - J\cos(k))\end{pmatrix}\underbrace{\begin{pmatrix}c_k\\ c_{-k}^\dagger\end{pmatrix}}_{\Psi_k} + \text{const.} \tag{41b}$$

$$= \sum_k \Psi_k^\dagger\underbrace{\begin{pmatrix}+z_k & -iy_k\\ +iy_k & -z_k\end{pmatrix}}_{M_k}\Psi_k + \text{const.} \tag{41c}$$

Here we defined $y_k = J\sin(k)$ and $z_k = \Gamma - J\cos(k)$ as shorthands to make $M_k$ easier to manage.

## B.3 Bogoliubov Transformation

The final step is to perform a Bogoliubov transformation, which is a linear transformation that preserves the anticommutation relations of $c_k^\dagger, c_k$. In our case this transformation actually ends up being a rotation

and it is therefore useful to first express $M_k$ as something that is more geometrical in its nature

$$M_k = \varepsilon_k \begin{pmatrix} \cos(2\theta_k) & -i\sin(2\theta_k) \\ i\sin(2\theta_k) & -\cos(2\theta_k) \end{pmatrix} = \begin{pmatrix} \varepsilon_k & 0 \\ 0 & \varepsilon_k \end{pmatrix} \begin{pmatrix} \cos(2\theta_k) & i\sin(2\theta_k) \\ i\sin(2\theta_k) & \cos(2\theta_k) \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \tag{42}$$

where

$$\varepsilon_k = \sqrt{y_k^2 + z_k^2} = \sqrt{\Gamma^2 - 2\Gamma J\cos(k) + J^2}, \quad \sin(2\theta_k) = y_k/\varepsilon_k, \quad \cos(2\theta_k) = z_k/\varepsilon_k. \tag{43}$$

Now $M_k$ can be viewed as a flip along the second axis, followed by a rotation of $2\theta_k$ and finally a scaling of $\varepsilon_k$. The unitary transformation $U_k$ that we will use to diagonalize $M_k$ is

$$U_k = \begin{pmatrix} \cos(\theta_k) & i\sin(\theta_k) \\ i\sin(\theta_k) & \cos(\theta_k) \end{pmatrix} \quad \text{with the property} \quad U_k \begin{pmatrix} \cos(\phi_k) \\ i\sin(\phi_k) \end{pmatrix} = \begin{pmatrix} \cos(\theta_k + \phi_k) \\ i\sin(\theta_k + \phi_k) \end{pmatrix} \tag{44}$$

for any $\phi_k$. We can now diagonalize $M_k$ and it is even possible to visualize it by viewing the matrices as manipulations in the complex plane

$$U_k^\dagger M_k U_k = \begin{pmatrix} \cos(\theta_k) & i\sin(\theta_k) \\ i\sin(\theta_k) & \cos(\theta_k) \end{pmatrix}^{-1} \begin{pmatrix} \varepsilon_k & 0 \\ 0 & \varepsilon_k \end{pmatrix} \begin{pmatrix} \cos(2\theta_k) & i\sin(2\theta_k) \\ i\sin(2\theta_k) & \cos(2\theta_k) \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} \cos(\theta_k) & i\sin(\theta_k) \\ i\sin(\theta_k) & \cos(\theta_k) \end{pmatrix} \tag{45a}$$

$$= \varepsilon_k \begin{pmatrix} \cos(\theta_k) & i\sin(\theta_k) \\ i\sin(\theta_k) & \cos(\theta_k) \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} \cos(\theta_k) & i\sin(\theta_k) \\ i\sin(\theta_k) & \cos(\theta_k) \end{pmatrix} \tag{45b}$$

$$= \begin{pmatrix} \varepsilon_k & 0 \\ 0 & -\varepsilon_k \end{pmatrix}. \tag{45c}$$

The Bogoliubov fermion operators for this transformation are

$$U_k^\dagger \Psi_k = \begin{pmatrix} \cos(\theta_k)c_k - i\sin(\theta_k)c_{-k}^\dagger \\ i\sin(\theta_k)c_{-k} + \cos(\theta_k)c_k^\dagger \end{pmatrix} = \begin{pmatrix} \eta_k \\ \eta_{-k}^\dagger \end{pmatrix}, \tag{46}$$

which follow the anticommutation relations of normal fermion operators. The Hamiltonian now assumes its final form

$$H = \sum_k (U_k^\dagger \Psi_k)^\dagger (U_K^\dagger M_k U_k)(U_k^\dagger \Psi_k) + \text{const.} \tag{47a}$$

$$= \sum_k \begin{pmatrix} \eta_k^\dagger & \eta_{-k} \end{pmatrix} \begin{pmatrix} \varepsilon_k & 0 \\ 0 & -\varepsilon_k \end{pmatrix} \begin{pmatrix} \eta_k \\ \eta_{-k}^\dagger \end{pmatrix} + \text{const.} \tag{47b}$$

$$= \sum_k (\varepsilon_k \eta_k^\dagger \eta_k - \varepsilon_k \eta_{-k}\eta_{-k}^\dagger) + \text{const.} \tag{47c}$$

$$= \sum_{k>0} \omega_k (\eta_k^\dagger \eta_k + \eta_{-k}^\dagger \eta_{-k}) + \text{const.} \quad \text{with} \quad \omega_k = 2\varepsilon_k \tag{47d}$$

$$= \sum_{k>0} H_k + \text{const.} \tag{47e}$$

Every $H_k$ has its own three-level system with four states: the ground state $|g_k\rangle \propto \eta_k\eta_{-k}|0\rangle$, the two excited equal-energy states $|e_{\pm k}\rangle \propto \eta_{\pm k}^\dagger \eta_{\mp k}|0\rangle$, and the highest energy state $|s_k\rangle \propto \eta_k^\dagger \eta_{-k}^\dagger |0\rangle$. Writing out the full forms of $\eta_{\pm k}^\dagger$ and $\eta_{\pm k}$ and normalizing we find the ground state and the most excited state in full are

$$|g_k\rangle = (\cos(\theta_k) + i\sin(\theta_k)c_k^\dagger c_{-k}^\dagger)|0\rangle \quad \text{and} \quad |s_k\rangle = (i\sin(\theta_k) + \cos(\theta_k)c_k^\dagger c_{-k}^\dagger)|0\rangle\,. \tag{48}$$

Unsurprisingly, they can also be expressed via each other through $|g_k\rangle = \eta_k\eta_{-k}|s_k\rangle$ and $|s_k\rangle = \eta_k^\dagger \eta_{-k}^\dagger |e_k\rangle$. The other excited states $|e_{\pm k}\rangle$ can be calculated similarly, but they will not be used by us and are therefore left out.

## B.4 Quantum Quench

Now we are ready to quench the system. This is done by starting out with the Hamiltonian $H'$ that has the parameters $J'$ and $\Gamma'$. At this point the system will be in its ground state $|g'\rangle = \prod_{k>0}(\cos(\theta'_k) +$

$i\sin(\theta'_k)c^\dagger_k c^\dagger_{-k})\,|0\rangle$. We then suddenly quench the system such that the Hamiltonian instantly changes to $H$ with parameters $J$ and $\Gamma$ (we do $H' \to H$ instead of $H \to H'$ to ease some of the later notation). Because the system was still in the state $|g'\rangle$, the system will now become dynamic and we can therefore begin calculating the Loschmidt amplitude

$$\mathcal{G}(t) = \langle g'|e^{-iHt}|g'\rangle = \prod_{k>0} \langle g'_k|e^{-iH_k t}|g'_k\rangle = \prod_{k>0} \mathcal{G}_k(t). \tag{49}$$

As $|g'_k\rangle$ is not an eigenstate of $H_k$ we have to translate it into a superposition of $|g_k\rangle$ and $|s_k\rangle$. This is possible because $|g'_k\rangle$ and $|s'_k\rangle$ have zero total momentum, so they both live in the subspace spanned by $|0\rangle$ and $c^\dagger_k c^\dagger_{-k}\,|0\rangle$. We can therefore work in the basis $(|0\rangle\,, c^\dagger_k c^\dagger_{-k}\,|0\rangle)$ and perform the following manipulations

$$\begin{aligned}
\begin{pmatrix}|g'_k\rangle & |s'_k\rangle\end{pmatrix} &= \begin{pmatrix}\cos(\theta'_k) & i\sin(\theta'_k)\\ i\sin(\theta'_k) & \cos(\theta'_k)\end{pmatrix} && \text{(50a)}\\
&= \begin{pmatrix}\cos(\phi_k) & i\sin(\phi_k)\\ i\sin(\phi_k) & \cos(\phi_k)\end{pmatrix}\begin{pmatrix}\cos(\theta_k) & i\sin(\theta_k)\\ i\sin(\theta_k) & \cos(\theta_k)\end{pmatrix} \quad \text{with} \quad \phi_k = \theta'_k - \theta_k && \text{(50b)}\\
&= \begin{pmatrix}\cos(\phi_k)\cos(\theta_k) + i\sin(\phi_k)i\sin(\theta_k) & i\sin(\phi_k)\cos(\theta_k) + \cos(\phi_k)i\sin(\theta_k)\\ \cos(\phi_k)i\sin(\theta_k) + i\sin(\phi_k)\cos(\theta_k) & i\sin(\phi_k)i\sin(\theta_k) + \cos(\phi_k)\cos(\theta_k)\end{pmatrix} && \text{(50c)}\\
&= \begin{pmatrix}\cos(\phi_k)\,|g_k\rangle + i\sin(\phi_k)\,|s_k\rangle & i\sin(\phi_k)\,|g_k\rangle + \cos(\phi_k)\,|s_k\rangle\end{pmatrix}. && \text{(50d)}
\end{aligned}$$

We see $|g'_k\rangle = \cos(\phi_k)\,|g_k\rangle + i\sin(\phi_k)\,|s_k\rangle$ with $\phi_k = \theta'_k - \theta_k$ and it is now a simple matter to compute $\mathcal{G}_k(t)$ by exploiting $\eta^\dagger_{\pm k}\eta_{\pm k}\,|g_k\rangle = 0\,|g_k\rangle$ and $\eta^\dagger_{\pm k}\eta_{\pm k}\,|s_k\rangle = 1\,|s_k\rangle$:

$$\begin{aligned}
\mathcal{G}_k(t) &= \langle g'_k|e^{-iH_k t}|g'_k\rangle && \text{(51a)}\\
&= \cos^2(\phi_k)\,\langle g_k|e^{-i\omega_k t(\eta^\dagger_k\eta_k + \eta^\dagger_{-k}\eta_{-k})}|g_k\rangle + i\sin^2(\phi_k)\,\langle s_k|e^{-i\omega_k t(\eta^\dagger_k\eta_k + \eta^\dagger_{-k}\eta_{-k})}|s_k\rangle && \text{(51b)}\\
&= \cos^2(\phi_k) + i\sin^2(\phi_k)e^{-i2\omega_k t}. && \text{(51c)}
\end{aligned}$$

Continuing, we use this result to additionally calculate the Loschmidt echo for all momentum $k$

$$\begin{aligned}
\mathcal{L}_k(t) &= |\mathcal{G}_k(t)|^2 && \text{(52a)}\\
&= (\cos^2(\phi_k) + i\sin^2(\phi_k)e^{-i2\omega_k t})(\cos^2(\phi_k) - i\sin^2(\phi_k)e^{i2\omega_k t}) && \text{(52b)}\\
&= \cos^4(\phi_k) + 2\cos^2(\phi_k)\sin^2(\phi_k)\sin(2\omega_k t) + \sin^4(\phi_k) && \text{(52c)}\\
&= \cos^4(\phi_k) + 2\cos^2(\phi_k)\sin^2(\phi_k)(1 - 2\sin^2(\omega_k t)) + \sin^4(\phi_k) && \text{(52d)}\\
&= 1 - \sin^2(2\phi_k)\sin^2(\omega_k t), && \text{(52e)}
\end{aligned}$$

from which we get the rate function

$$\begin{aligned}
\lambda(t) &= -\frac{1}{L}\log\left(\prod_{k>0}\mathcal{L}_k(t)\right) && \text{(53)}\\
&= -\frac{1}{L}\sum_{k>0}\log\big(1 - \sin^2(2\phi_k)\sin^2(\omega_k t)\big) && \text{(54)}\\
&\approx -\frac{1}{2\pi}\int_0^\pi \log\big(1 - \sin^2(2\phi_k)\sin^2(\omega_k t)\big)\,\mathrm{d}k\,. && \text{(55)}
\end{aligned}$$

For explicit computation we can also write

$$\sin(2\phi_k) = \sin(2\theta'_k)\cos(2\theta_k) - \cos(2\theta'_k)\sin(2\theta_k) = \frac{(J'\Gamma - J\Gamma')\sin(k)}{\varepsilon'_k\varepsilon_k}. \tag{56}$$

## B.5 Dynamical Quantum Phase Transitions

We see that a DQPT will only happen at time $t$ if $\sin^2(2\phi_k)\sin^2(\omega_k t) = 1$ for some $k$. This can only happen if $\sin^2(2\phi_k) = 1$, which is independent of $t$, so it is best to find $k$ and thereafter get $t$ from

$\sin^2(\omega_k t) = 1$. Because $\sin(k)$ is not bijective on $k \in (0, \pi)$ it is not possible to use Eq. 56, so instead we look at the equivalent condition $\cos(2\phi_k) = 0$ which yields

$$0 = \cos(2\phi_k) = \cos(2\theta'_k)\cos(2\theta_k) + \sin(2\theta'_k)\sin(2\theta_k) = \frac{(J'\Gamma + J\Gamma')\cos(k) - (J'J + \Gamma'\Gamma)}{\varepsilon'_k \varepsilon_k}, \tag{57}$$

so

$$\cos(k^*) = \frac{J'J + \Gamma'\Gamma}{J'\Gamma + J\Gamma'} = \frac{1 + (\Gamma'/J')(\Gamma/J)}{(\Gamma'/J') + (\Gamma/J)}. \tag{58}$$

This expression also serves as a condition for whether a DQPT will occur in the rate function for any given $J', J, \Gamma', \Gamma$ as we require $|\cos(k^*)| < 1$. It is possible to find that this is equivalent to DQPTs happening if and only if $|\Gamma'/J'| < 1 < |\Gamma/J|$ or visa versa. The last part is to find all times $t_n$ where a DQPT appears, which is easily done

$$t_n = \frac{\pi(n + 1/2)}{\omega_{k^*}} \quad \text{where} \quad \omega_{k^*} = 2\sqrt{(J + \Gamma)(J - \Gamma)\frac{(\Gamma'/J') - (\Gamma/J)}{(\Gamma'/J') + (\Gamma/J)}}. \tag{59}$$

# References


[1] Karol Gregor et al. "DRAW: A Recurrent Neural Network For Image Generation". In: *arXiv:1502.04623 [cs]* (May 2015). arXiv: 1502.04623. URL: http://arxiv.org/abs/1502.04623.

[2] N. S. Kumar, Mahathi Amencherla, and Manu George Vimal. "Emotion Recognition in Sentences - A Recurrent Neural Network Approach". en. In: *Computational Intelligence in Data Science*. Ed. by Aravindan Chandrabose et al. IFIP Advances in Information and Communication Technology. Cham: Springer International Publishing, 2020, pp. 3–15. ISBN: 9783030634674. DOI: 10.1007/978-3-030-63467-4_1.

[3] Anasua Chatterjee et al. "Semiconductor Qubits In Practice". In: *Nature Reviews Physics* 3.3 (Mar. 2021). arXiv: 2005.06564, pp. 157–177. ISSN: 2522-5820. DOI: 10.1038/s42254-021-00283-9. URL: http://arxiv.org/abs/2005.06564 (visited on 11/25/2021).

[4] Markus Heyl. "Dynamical quantum phase transitions: A brief survey". In: *EPL (Europhysics Letters)* 125.2 (Feb. 2019), p. 26001. ISSN: 1286-4854. DOI: 10.1209/0295-5075/125/26001. URL: https://iopscience.iop.org/article/10.1209/0295-5075/125/26001.

[5] Lior Pachter. *What is principal component analysis?* May 2014. URL: https://liorpachter.wordpress.com/2014/05/26/what-is-principal-component-analysis/.

[6] Roman Cheplyaka. *Explained variance in PCA*. Dec. 2017. URL: https://liorpachter.wordpress.com/2014/05/26/what-is-principal-component-analysis/.

[7] Michael Aaron Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/.

[8] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]* (Jan. 2017). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[9] Federico Fedele. "Spin interactions within a two-dimensional array of GaAs double dots". PhD thesis. Niels Bohr Institute: University of Copenhagen, Feb. 2020.

[10] A. C. Johnson et al. "Singlet-triplet spin blockade and charge sensing in a few-electron double quantum dot". In: *Physical Review B* 72.16 (Oct. 2005). arXiv: cond-mat/0410679, p. 165308. ISSN: 1098-0121, 1550-235X. DOI: 10.1103/PhysRevB.72.165308. URL: http://arxiv.org/abs/cond-mat/0410679.

[11] Philip Krantz et al. "A Quantum Engineer's Guide to Superconducting Qubits". In: *Applied Physics Reviews* 6.2 (June 2019). arXiv: 1904.06560, p. 021318. ISSN: 1931-9401. DOI: 10.1063/1.5089550. URL: http://arxiv.org/abs/1904.06560 (visited on 12/08/2021).

[12] Markus Heyl. "Dynamical quantum phase transitions: a review". In: *Reports on Progress in Physics* 81.5 (May 2018), p. 054001. ISSN: 0034-4885, 1361-6633. DOI: 10.1088/1361-6633/aaaf9a. URL: https://iopscience.iop.org/article/10.1088/1361-6633/aaaf9a.

[13] Glen Bigan Mbeng, Angelo Russomanno, and Giuseppe E. Santoro. "The quantum Ising chain for beginners". In: *arXiv:2009.09208 [cond-mat, physics:quant-ph]* (Sept. 2020). arXiv: 2009.09208. URL: http://arxiv.org/abs/2009.09208.

[14] Neil J. Robinson, Isaac Pérez Castillo, and Edgar Guzmán-González. "Quantum quench in a driven Ising chain". In: *Physical Review B* 103.14 (Apr. 2021). arXiv: 2011.14345, p. L140407. ISSN: 2469-9950, 2469-9969. DOI: 10.1103/PhysRevB.103.L140407. URL: http://arxiv.org/abs/2011.14345.