KetGPT: Generating Quantum Circuits Using Transformers

by

Boran Apak

To obtain the degree of Master of Science in
Applied Physics at the Delft University of Technology.
To be publicly defended on November 30th, 2023 at 14:00

Project duration:

January 7, 2023 - November 30, 2023

Thesis committee:

Dr. S. Feld (Supervisor)
Dr. F. Sebastiano
Dr. G. Vardoyan

Student number:

4552342

Preface and Acknowledgements

I have known all my life that I wanted to be a physicist. I was fascinated by, and desperately wanted to learn about the inner workings of nature. That is why I was devastated when I was told that I was not allowed to go into that direction in high school, and that I should focus on becoming an economist instead. Stubborn as I am, I took physics as an extra course while I continued my high school career. Because this extra course was my favourite course in the curriculum, I realised that I wanted to do everything in my power to be able to go into the field of physics anyway. Despite my teachers telling me that I shouldn't, I bought some mathematics books and started preparing for an exam I could take outside of school, that would allow me to pursue the physics degree I had always wanted. Years later, I am now writing my master thesis in applied physics, and I think nothing could make the younger version of me happier than to hear that after a lot of hard work, his dream to become a physicist came true.

Many people have been instrumental in achieving this dream, and I would like to take a moment to acknowledge them.

I want to express my gratitude to my supervisors Sebastian Feld, Medina Bandic and Aritra Sarkar for the opportunity to work on this project, their guidance, and our many interesting discussions during my thesis.

I am very grateful to Dr. Fabio Sebastiano and Dr. Gayane Vardoyan for agreeing to be a part of my thesis committee.

I also thank the members of the QML group, from whom I've learned a lot, with whom I've had many nice chats and with whom I've eaten a lot of cake during our weekly group meetings.

Furthermore, I want to thank Pien Abbink, for being my study buddy during my thesis project, and for always calming me down when I was stressing out over yet another small bump I encountered during this time.

Lastly, I want to thank my family and friends that have lovingly supported me during my studies. I couldn't have done it without you by my side.

Abstract

The goal of this thesis is expanding quantum algorithm datasets to enhance our capability to benchmark quantum systems and to open up possibilities for using machine learning techniques in quantum circuit mapping. Both of these areas are currently hindered by the lack of a wide range of useful quantum algorithms. To solve this problem, KetGPT is presented, a model that uses the revolutionary transformer machine learning architecture to generate synthetic, yet realistic looking, quantum circuits. By visual inspection, KetGPT generated circuits are easily distinguishable from random circuits, and show desirable qualities such as structure and human-like programming factors including applying gates in the order of ascending qubits. Consequently, they might be more suitable for certain tasks like benchmarking and training a reinforcement learning compiler. In an attempt to quantify the quality of circuits generated by KetGPT, a separate transformer classifier model was trained on the task of classifying the synthetic circuits generated by KetGPT as either real circuits, or as random circuits. However, although this classifier might capture realistic features of quantum circuits, the classifier has not been unambiguously proven to be reliable, and can therefore not be used as a standalone tool to determine the quality of KetGPT generated quantum circuits. Nevertheless, KetGPT and the transformer classifier are novel, promising approaches in an attempt to expand quantum algorithm datasets.

Contents

1 Introduction

The emergence of transformer models [1], that are part of the technology behind ChatGPT [2], has caused a paradigm shift in the field of natural language processing. These models are used for realistic text and code generation [3, 4] and they achieve impressive performance on these tasks by capturing important information about the structure of sequences of data, and make it possible to use this information to generate new synthetic, but realistic looking, data.

Simultaneously, another potentially groundbreaking technology is being developed: quantum computers. Quantum computers are a promising type of computers that, in theory, can solve certain problems faster than classical computers can [5]. To solve those problems, quantum algorithms need to be carried out on those quantum computers using quantum circuits, that are defined using quantum code like Quantum Assembly (QASM) [6]. However, at the moment there are only a handful of quantum algorithms that are known to solve a useful problem in a way that is faster than with classical algorithms, which makes it challenging to benchmark those quantum computers and related software. Furthermore, there are machine learning applications where there is a need for a large dataset of quantum algorithms [7, 8].

In an attempt to address these problems in quantum computing, and inspired by the paradigm shift in natural language processing, the **problem statement** of this thesis is as follows:

"Can transformer models be used to generate realistic-looking quantum circuits to expand quantum algorithm datasets?"

To resolve this question, this thesis introduces KetGPT: a transformer model that is capable of generating realistic-looking quantum circuits. Furthermore, a method to determine the quality of the generated QASM code is proposed, using a different transformer model specifically designed for this task.

The remainder of the thesis is structured as follows: in Section 2, background on quantum algorithms is provided in order to acquaint the reader with quantum computing. In Section 3, transformer models are introduced, the highly influential type of machine learning models that excel in generative tasks like generating data. Section 4 introduces the main contribution of this thesis: KetGPT, a transformer model specifically designed to generate data that is useful for quantum computing purposes. It describes how to generate QASM files using these transformer models, and in this section a method to quantify how realistic these QASM files are is proposed. In Section 5 the results are presented and the generated QASM files are examined. Section 6 contains a discussion of the presented results, and a conclusion is presented in Section 7. Section 8 elaborates on suggestions for future work. Ultimately, supplementary information like the Python code, the dataset that was used, and examples of generated QASM files are brought forward in Section 9.

2 Quantum Algorithms and Computation

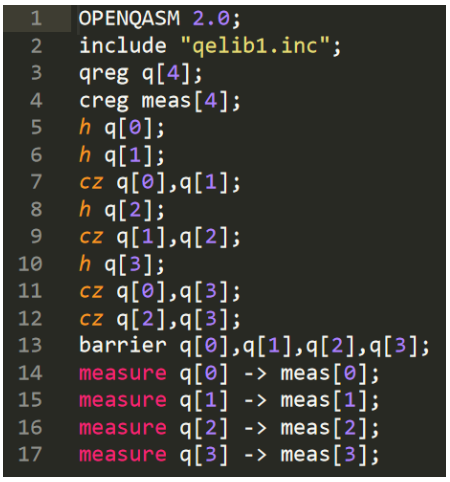
In general, algorithms are sequences of operations that transform an input to an output, often in the context of achieving a certain goal [9]. For example in classical computing if we want to find a certain number in a list of numbers, an algorithm to find that specific number could be: start at the beginning of the list and check if that number is the number you are looking for, if not: check the next one until you find the desired number. The output of this algorithm would be the position of the number in the list. As this example shows, in the context of algorithms we usually have a problem, an idea to solve this problem, and a structured description of how to solve this problem. This structured description of how to solve problems can be in the form of natural language, but can also be formalised and written in the form of code. Quantum algorithms are a certain class of algorithms that can be implemented on quantum computers using quantum code, like Quantum assembly (QASM) [6]. Some of these algorithms are particularly interesting since they could solve some problems faster than classical computers can solve these problems [5]. In this section, a short overview of current algorithms will be presented, followed by a general description of the systems that are needed to execute these algorithms, and finally some comments on benchmarking these systems.

2.1 Overview of Quantum Algorithms

"The Quantum Algorithm Zoo" [10] is a catalog of quantum algorithms that gives a rough idea of how many quantum algorithms exist that have a so called "quantum speedup", i.e. quantum algorithms that can solve problems in less time than the best known classical algorithms can solve these problems [11]. At the time of writing there are 64 entries, and although this number itself is rather arbitrary, since for example some machine learning applications are grouped and counted as one, the main takeaway from this number is that it signifies that there are only a limited number of quantum algorithms that provide a quantum speedup. Furthermore not all of these algorithms are equally interesting, since some algorithms are only going to be used for niche domains, having a relatively small societal impact, so it can be said that quantum algorithms that are expected to be widely used in practice are quite scarce. Some quantum algorithms that are expected to have a big societal impact are for example [5]: Grover's search algorithm [12], Shor's factoring algorithm [13] and the quantum algorithm for solving linear systems of equations (commonly called HHL) [14]. Some reasons why the amount of useful algorithms are scarce could be: classical algorithm designers need specialised education to be able to design quantum algorithms, quantum mechanics is counter-intuitive for humans and the applications can be very technical and specialised [15].

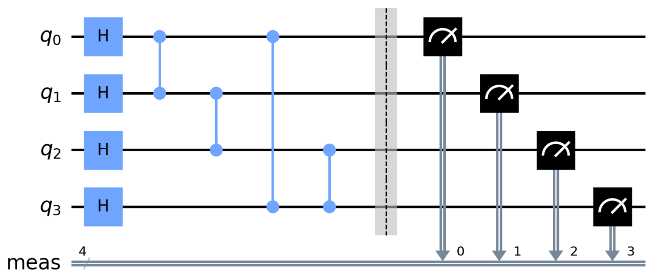
2.2 QASM Files and Quantum Circuits

As mentioned in the introduction to this section, Quantum algorithms are conventionally encoded using Quantum assembly (QASM). An example of a QASM file, containing QASM code is shown in Figure 1a. We can also graphically depict the circuit for this QASM code, this circuit is shown in Figure 1b.



```
 1  OPENQASM 2.0;
 2  include "qelib1.inc";
 3  qreg q[4];
 4  creg meas[4];
 5  h q[0];
 6  h q[1];
 7  cz q[0],q[1];
 8  h q[2];
 9  cz q[1],q[2];
10  h q[3];
11  cz q[0],q[3];
12  cz q[2],q[3];
13  barrier q[0],q[1],q[2],q[3];
14  measure q[0] -> meas[0];
15  measure q[1] -> meas[1];
16  measure q[2] -> meas[2];
17  measure q[3] -> meas[3];
```

(a) An example of QASM code.



(b) Corresponding circuit.

Figure 1: QASM code with corresponding circuit.

Lines in the QASM file from Figure 1a correspond to an operation: line 5 dictates that a hadamard gate [16] should be performed on qubit 0, and line 7 means that a controlled phase gate [16] is performed between qubit 0 and qubit 1.

2.3 Implementing Quantum Algorithms

To run a quantum algorithm on a quantum computer, the QASM representation of an algorithm needs to be converted to instructions that a quantum computer can interpret. This is non trivial due to several factors, including:

- Different quantum hardware implementations can have a different native gate set, which are the operations that are natively supported by the quantum hardware device. For example when there is a cx gate in the QASM code, but cx is not in the device's native gate set, the cx needs to be converted to for example a cz gate (possibly by performing extra hadamards on the target qubit) to make sure this algorithm is able to be executed on this device. As long as the quantum hardware has a universal gate set, meaning that the gates from native gate set of the quantum hardware can perform any quantum computation, this conversion is realisable [17], but might increase the amount of gates which need to be executed to perform the computation.

- Different qubit topologies, which put limits on qubit connectivities and therefore determines between which qubits it is possible to perform 2-qubit gates.
- And lastly, there are multiple different ways to assign initial qubit numbers (if you have two physical qubits for example, you can choose which one you call "qubit 1"). This choice is often interwoven with qubit topology, since it makes sense to choose them in such a way that qubits that are supposed to interact are physically able to interact as much as possible, to prevent unnecessary SWAP gates.

The process of modifying quantum circuits to meet the physical constraints of a quantum device is called "quantum circuit mapping" [16, 18], and is an essential part of the compilation process. There are many different ways to do quantum circuit mapping, and to determine which method performs the best, benchmarking suites have been proposed [19–23].

2.4 Random Circuits for Quantum Circuit Mapping

Random circuits are composed of randomly chosen operations, in contrast to real circuits where operations are performed with a specific purpose in mind. Because of the fact that there are not so many useful algorithms, random circuits are deployed in various tasks related to quantum circuit mapping. Some benchmarking suites contain random circuits as part of their benchmark [19, 23].

Random circuits are also often used in machine learning tasks related to quantum circuit mapping [8, 24]. They are both used as training data, and for testing how well the machine learning models perform.

3 Transformers

Transformer models are popular machine learning models that excel at capturing dependencies in sequential data. This is why they have revolutionised natural language processing [1] and are used in, for example, code generation [4,25], and music generation [26]. In this chapter, a brief background on these models will be given and the reason why these models are promising for generating realistic-looking QASM code will be introduced.

3.1 Evolution of Natural Language Processing Models

Before transformers were introduced, convolutional neural networks (CNNs) [27], recurrent neural networks (RNNs) [28] and long short-term memory networks (LSTMs) [29] were the standard models used for natural language processing tasks like generating text. But these models had multiple issues like struggling with long-range dependencies and lack of parallelisability [1]. To resolve this, researchers developed the transformer architecture, first published in the highly influential paper "Attention is All you Need" [1]. Transformer models differentiate themselves from earlier models because they are parallelisable (and are therefore more suitable for training on large datasets) and they are able to capture longer-range dependencies.

3.2 Components of a Transformer

Now that the advantages of transformer models have been discussed, we will further specify how they are structured. First, all the individual components of a transformer architecture will be discussed following the path of data flow, then in Section 3.3 the full model architecture will be discussed in its entirety.

3.2.1 Tokenisation of Dataset

In computers, text is represented as a so-called "string". The calculations we are going to use, however, work with numbers instead of strings (a list of characters), so we need to convert those strings to numbers. These numbers are called "tokens" and this process is called "tokenisation". Although this part of the process is not strictly part of the transformer architecture as defined in [1] (rather it is part of dataset preparation), it is essential in understanding the flow of information through the transformer network and is therefore included in this section. An example of this process is shown in Table 1.

Table 1: Tokenisation Example. Text (in the form of QASM operations) is provided, and each statement (a line of QASM code) is converted to a number. The number that is assigned to each statement does not have an intuitive meaning, rather, it just depends on the way our tokenisation algorithm has ordered its vocabulary. Tokenising a sequence of statements will create a list of numbers.

QASM Operation	**Tokenized Sequence**
`h q[0];`	[9]
`cx q[0],q[1];`	[55]
`swap q[1],q[2];`	[12]

`h q[0];`
`cx q[0],q[1];`
	[9, 55]

`h q[0];`
`cx q[0],q[1];`
`swap q[1],q[2];`
	[9, 55, 12]

One can imagine that there are multiple ways to divide strings up into numbers, for example, using the approach shown in Table 1. But we can also convert every character to a number, e.g.

```
h q[0];
```

could also be converted to the 7 integers

```
[8, 37, 17, 38, 27, 39, 40]
```

for the 7 characters including whitespace, instead of just

```
[9]
```

So to characterize the tokenisation process, we need to have a system for dividing up a sequence into parts, and we need to have a "dictionary" to determine what number we associate with each possible part of a sequence we can encounter using our system for dividing up sequences.

In the next sections, we will use the last row of Table 1 to further clarify the information flow within the corresponding component.

3.2.2 Word Embeddings

In the last example in Table 1 we can see that the sequence of QASM operations has been converted to a sequence of integers. After this step, we convert every integer in the sequence into a real-valued vector, to obtain our word embeddings. So in the aforementioned example, [9,55,12] will be converted into a real-valued array of dimensions $3 \times dim$, where 3 is the length of the sequence, and dim is the dimension of the vector. The dimension of this vector that represents an individual QASM operation is a free parameter of the model and can be set to any integer, where a higher dimension makes the model potentially better, but is more costly in memory and training time. This vector representation is used to better capture the "meaning" of a word since, for example, the vectors representing "h q[0];" and "h q[1];" can be more similar to than, say, the vectors representing "h q[0];" and "swap q[42], q[23];". This process of converting a token into a vector is done using a Feed Forward Neural Network, so this part of the model is the first component that we have encountered that is trainable. What a Feed Forward Neural Network is will be detailed in Section 3.2.4. At the end of this process, we have for our example in Table 1 a sequence of 3 vectors, where the 9, 55 and 12 are each represented by a vector of a chosen dimension, so we end up with an array of real numbers with size $3 \times dim$ that represents our original QASM code.

3.2.3 Positional Encoding

The order of lines of QASM code matters: for example `h q[0];` and then `cx q[0],q[1];` will result in a different outcome than by first applying `cx q[0],q[1];` and then `h q[0];`. However, transformer models don't innately have a way to capture this positional information, so positional information has to be added to the data. This is done using a process called positional encoding, which is graphically depicted in Figure 2. We take the sequence of vectors that we created in Section 3.2.2, and we add a sequence of vectors that signify the position of the individual QASM statements. The values of the elements of these positional encoding vectors are trainable parameters of the model, and are therefore determined during the training process.

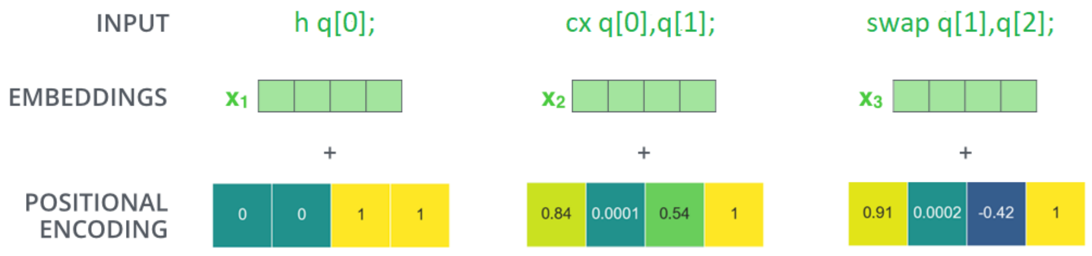


Figure 2: Graphical example of positional encoding. A positional encoding vector is added to each word embedding vector. The values of the positional encoding vector depend on the position of the input in the sequence. Adapted from [30]

In the example from Table 1, after the positional encoding step we would still have an array of real numbers with size $3 \times dim$, as this step does not change the data shape. But for each of the 3 input vectors, a vector has been added to the dim dimensional vector that helps our network encode the positional information of the QASM statement, so we end up with the same size array with different numbers.

3.2.4 Feed Forward Neural Network

Neural Networks play a key role in various machine learning techniques and are one of the fundamental parts of transformer models. They consist of a series of layers that each perform a linear operation on the input followed by a (non-linear) activation function. An example of a Neural Network is given in Figure 3.

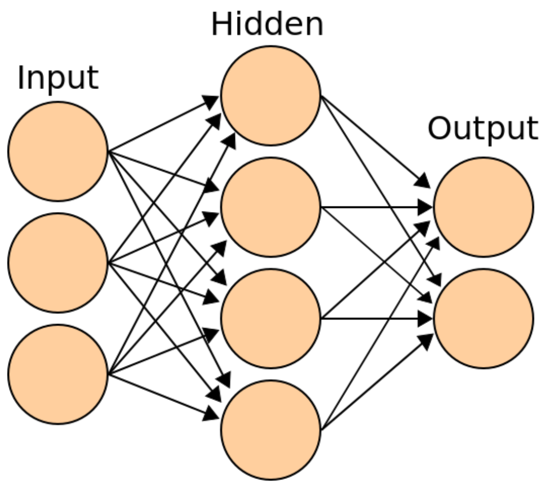


Figure 3: Schematic depiction of a Feed Forward Neural Network with 1 hidden layer [31].

In Figure 3, every circle represents a node. Every node is assigned a real number. The number in a node follows from the following equation:

$$\begin{aligned} h_1 &= Activation(\{w_{11} \cdot x_1\} + \{w_{12} \cdot x_2\} + \{w_{13} \cdot x_3\} + b_1) \\ h_k &= Activation(\{w_{k1} \cdot x_1\} + \{w_{k2} \cdot x_2\} + \{w_{k3} \cdot x_3\} + b_k), \end{aligned} \tag{1}$$

where h_1 is the value of node 1 in the hidden layer, x_1 is the value of node 1 in the layer before it, w_{12} the weight of the connection between node 2 in the layer before it and node 1 in the layer, and b_1 is some number that serves as a bias. Then a non-linear activation function (like softmax [32] or ReLu [33]) is applied, so that the network can capture complex non-linear patterns. In general, the value of node k will be a linear combination of the values of the nodes in the previous layer weighted by the corresponding weights, passed through an activation function. In the example from equation (1), it is assumed that the previous layer only contains 3 nodes, and in that case w_{13} is the last weight. But more generally, this pattern of multiplying every node in the previous layer with a weight is continued for the total amount of nodes in the previous layer. A Feed Forward Neural Network is fully defined by specifying the number of layers, the number of nodes in each layer, the weights of every connection between nodes of a layer and a previous layer, a bias per node and the activation function per layer.

To train such a network one initialises a network with the desired architecture, but with randomly initialised weights and biases, and then trains the network by repeatedly giving an input and adjusting those randomly initialised weights and biases to better match the output of the network with the expected output for that specific input. These weights and biases are adjusted using a method called Stochastic Gradient Descent [34]. The simplest form of this method can be described as follows: first define a loss function that determines how well the model results fit your expected results (for example, taking the mean square error), then find the partial derivative of the loss function to every weight and bias and update the weights and biases according to the following formula:

$$\mathbf{w} := \mathbf{w} - \alpha \nabla C(\mathbf{w}), \tag{2}$$

where $\mathbf{w}$ is a vector containing all weights and biases, C is your loss function and α is a parameter called the learning rate. The $-\nabla C(\mathbf{w})$ term determines in which direction you need to adjust the weights and biases vector to lower the loss function, and the learning rate α determines how drastically you want to adjust the weights and biases in that direction. One of the reasons machine learning has been so successful is because of the backpropagation algorithm, which is an algorithm that allows us to efficiently compute the gradient needed for a Stochastic Gradient Descent step [35].

3.2.5 Self-Attention

Self-attention is a mechanism that helps a transformer understand the relation between words, and is the main innovation in transformer models. For example, in the sentence "The computer executes the program, because it is told to", for humans it is easy to know that "it" refers to the computer and not to the program, but for computers, this is not so obvious. A self-attention component helps transformers make this connection.

The input to the attention mechanism consists of queries, keys and values. Each token in the input sequence corresponds to one query and key vector with dimension d_k and a value vector with dimension d_v, but for computational purposes, the queries, keys and values for all tokens are packed into respectively, a matrix Q, K and V. These matrices can be derived in the manner graphically depicted in Figure 4:

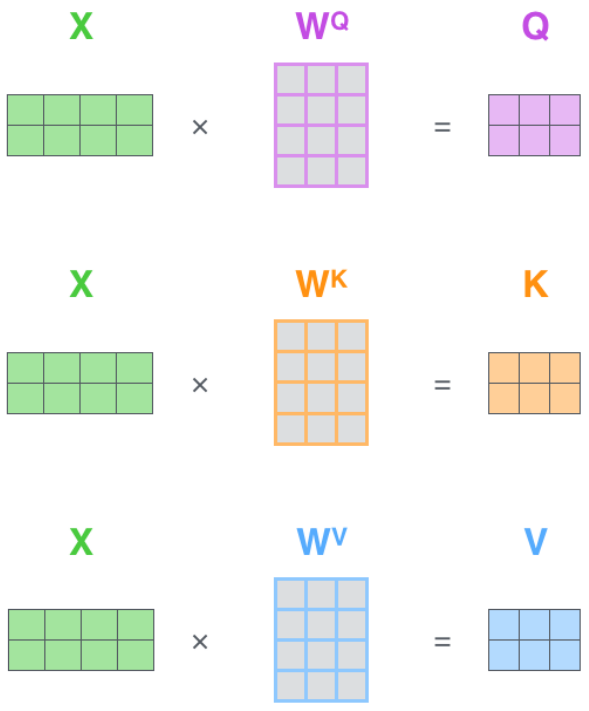


Figure 4: Every row in the X matrix corresponds to a token in the input sentence. In this figure the sequence length is 2 tokens. The W^Q, W^K and W^V matrices are learned in the process of training the model [30]

Then, the main equation describing the attention process is the following: [1]

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{3}$$

where softmax is the softmax function [32] and K^T is the transpose of the K matrix.

The intuition behind this equation is that the QK^T term describes a dot product between the queries and the keys, to determine their “inter-relation”. Then this information is used to create an attention matrix that can be interpreted as being similar to a correlation matrix as it determines for each element how strong its bond is to each other element, but instead of it being a correlation that is between -1 and 1, it is in the form of a probability distribution, so with values between 0 and 1. The $\sqrt{d_k}$ scaling factor is there to get a more dimension-independent dot product, and that for technical reasons makes it easier to train the network [1]. Then, after multiplying this attention matrix with V, we get our final result which is our original matrix V with added information about the inter-relations between the queries and the keys because, for example, values that had a very low score in the attention matrix are drowned out (because that corresponding attention matrix element is close to 0). To give a more concrete example: if information in the sentence “The computer executes the program, because it is told to” that was alluded to earlier, is encoded in 3 matrices Q, K and V. Then Attention(Q, K, V) gives us a matrix that encodes this sentence with information about the inter-relations between the words (for example the information that “it” refers to the computer and not the program).

3.2.6 Multi-Head Attention

Since words are related in multiple ways (e.g., syntactically or semantically), a transformer model uses multiple attention heads that act on the input at the same time, to get a richer representation of the relations within the sentence. Mathematically this is implemented by simply repeating the process in the previous section (Section 3.2.5), but for various weight matrices $\mathrm{W^Q}$, $\mathrm{W^K}$ and $\mathrm{W^V}$. We end up with multiple matrices that encode our original sentence with information about the inter-relations of the words in that sentence. However, we want to end up with only one final representation of this sentence instead of a collection of various representations. To this end we concatenate all matrices, and we multiply the result with a matrix $\mathrm{W^O}$ so that the resulting matrix has our expected dimensions again (that are independent of the number of heads we use). The values of the elements of $\mathrm{W^O}$ are trainable during the training process. This gives our model the desired ability to learn many ways in which the words in our sentence are related to each other, and even lets it give the different ways words are related to different weights [1].

3.2.7 Masked Multi-Head Attention

In regular Multi-Head attention we allow every token in a sequence to attend to each other. However, when generating a sequence, we want to prevent leftward information flow and we want to preserve the auto-regressive property. To do so, in a Masked Multi-Head Attention block, matrix elements of the Q and K matrices that correspond to "illegal connections" are set to $-\infty$ just before the softmax in the attention process [1].

3.2.8 Residual Connections

Within a Machine learning model every block (like a Feed Forward Neural Network, or a Self-attention block) tries to learn how an input should best be converted to an output. Often this output however is close to the input, so then it is much more meaningful to learn the *difference* between the input and the output. On a more technical note, learning the difference between input and output also helps solve the vanishing gradient problem, making models like these train efficiently [36].

Schematically this procedure, which is called a "residual connection", comes down to the following

$$\text{output} = \text{Block}(\text{input}) + \text{input}, \tag{4}$$

where the Block() function is a Neural Network or a Self-attention block and + is elementwise-addition.

3.2.9 Layer Normalisation

The output of an attention block, or a Feed Forward Neural Network block can have very high element values or very low element values, depending on the input of that block. We normalise these outputs according to the following formula:

$$\text{LayerNorm}(x) = \gamma \left(\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta, \tag{5}$$

where x is the output of an attention or Feed Forward Neural Network block, μ and σ^2 are respectively the mean and variance of x, ϵ is a small number in the denominator for numerical stability, and γ and β are learnable parameters.

By normalising the outputs in this way, it has been empirically shown that networks train better [37].

3.2.10 Classification Layer

At the end of the model there is a Classification layer. This is just a Feed Forward Neural Network with one layer, where the amount of output nodes is the same as the total amount of possible outcomes of the model (e.g. every word in a dictionary). After this Neural Network layer an activation function (usually softmax [32]) is applied that makes sure that the output of the layer is a probability distribution. So after this classification layer we end up with a probability distribution over our “dictionary” of possible words to choose from.

3.3 Full Model Architecture

After discussing every important individual component in the transformer architecture, the full model architecture, depicted in Figure 5, can be discussed:

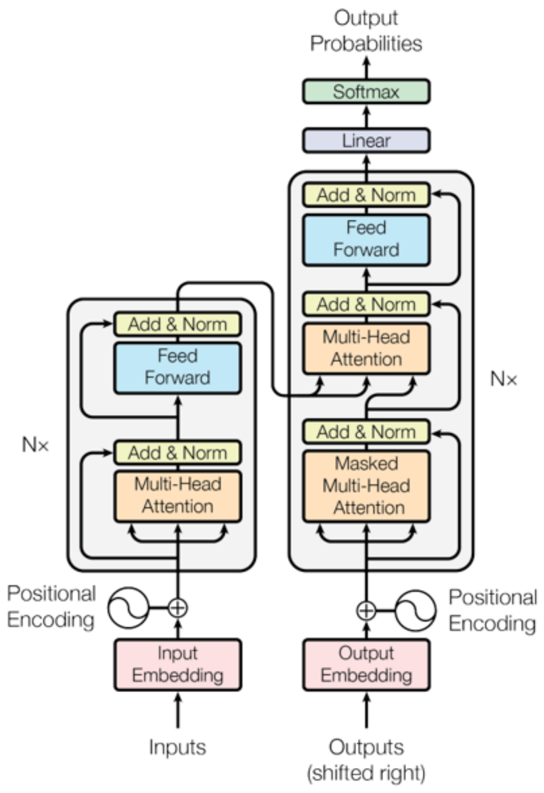


Figure 5: General transformer architecture [1]

We can discern two major parts in this architecture: the so-called “encoder” on the left side, and the “decoder” on the right side. This is the general transformer architecture, where both the encoder and decoder are present and they interact with each other, but different models use different architectures. For example, the GPT family of models use a decoder-only architecture, where there is only a decoder and no encoder [38]. Such an architecture looks like Figure 6:

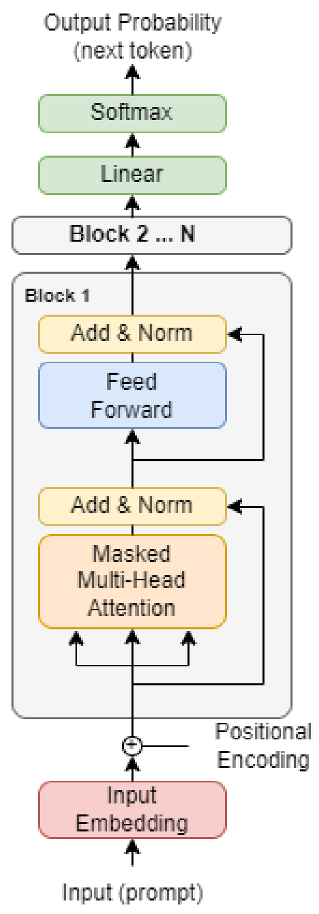


Figure 6: Decoder-only transformer architecture [39]

These decoder-only models are generally used for generation tasks like generating text [40] and generating computer code [25].

Similarly, there are also encoder-only models like the highly influential BERT model [41]. They look the same as the decoder-only model pictured in Figure 6, but they don't use Masked Multi-Head Attention. Rather they just use normal Multi-Head Attention, and therefore allow bidirectional (non-causal) information flow. These models are commonly used in sequence classification (sentiment analysis or spam detection for example) [42]. So for text generation tasks, one should employ a decoder-only transformer model, while for sequence classification tasks, one should employ an encoder-only transformer.

3.4 Information Flow Example

In this section we will exemplify the information flow within a transformer model. We will follow the last example from Table 1 again, and we will see what happens within a pass of a decoder-only transformer. We start with our input:

```
h q[0];
cx q[0],q[1];
swap q[1],q[2];
```

which we then tokenise (each valid QASM statement will be converted into a token) to get the following list of integers:

```
[9, 55, 12]
```

This list is now ready to be passed to the decoder-only transformer, as shown in Figure 6. The "input Embedding" step makes our list of integers into a list of real-valued vectors, which will be denoted with the letters a, b and c for the vectors corresponding to 9, 55 and 12 respectively.

```
[a, b, c]
```

The dimension of these vectors is a constant defined by the model. Afterwards, these vectors get modified in the positional encoding step so that they contain positional information in the positional encoding step. The vectors change but remain the same dimension. We denote these vectors with d, e and f representing the vectors a, b and c, respectively, but with added positional information:

```
[d, e, f]
```

Now, we arrive at the main point of interest: the Masked Multi-Head Attention block. We multiply our list of vectors (which can be represented by a matrix) with matrices W^{Q}, W^{K} and W^{V}, to get out matrices Q, K and V respectively. We set the values in the Q and K matrices that correspond to illegal connections to $-\infty$ and then we use these matrices according to the Attention formula described in equation (3), to get a matrix

Z_0

We repeat this process for the amount of "heads" we desire, to get matrices Z_0 , Z_1, etc. and we concatenate them and multiply that concatenated matrix with another weight matrix W^{O} to get a matrix

Z

Then we come to the Add & Norm block. In this block we add our matrix containing the vectors d, e and f to Z (element-wise addition) to get a matrix that represents our input sequence with information about relations between tokens. We normalise each row of this matrix (every row represents a token) using the LayerNorm formula described in equation (5). We call the resulting matrix

A

Each row of this matrix is then individually fed into a Feed Forward Neural Network. Then we get an Add & Norm block again, where we add A to the result and then normalise that matrix per row. Let's call this matrix

B

The process between the part after the positional encoding until this point is usually called a decoder block. We can repeat this process multiple times to get an even more complex representation of our original sequence.

After going through the desired amount of decoder blocks, the rows of the resulting matrix will individually go through the classification layer, which is one final linear layer, followed by a Softmax operation, to end up with a matrix where each row is a probability distribution over the vocabulary (all possible tokens).

3.5 Generating Text Using a Transformer

In generating text, we are only interested in the output that corresponds to the last token from the input sequence, so for example, to find the token that is most likely to follow our input sequence, we look at which element of the last row of the resulting matrix has the highest value. If we want to generate text or QASM code for example, we simply add the token we found to the input, go through the whole model again and repeat that process until we have generated the desired amount of tokens.

3.6 Training a Transformer Model

Now that we know how a transformer model works, we discuss how such a model can be trained.

We start with a training dataset that contains many data points (where every data point is a sequence of tokens). In this thesis a data point corresponds to one QASM file. One creates a batch of data points, which all individually travel through the model like in Section 3.4. Then for every token from every data point in the batch, we obtain a probability distribution over all tokens in our vocabulary, describing how likely it is for every token in the vocabulary to follow the input token, which is the end result of our model. To quantitatively assess our model performance we use a so-called "loss function" to compare the computed probability distribution for the next token in the sequence with what token was actually next in the sequence. Since choosing which token from a set of tokens is the one we are looking for is a classification problem, we use the Cross-Entropy loss function [43], which is the standard loss function for classification problems [38, 41]:

$$-\sum_{c=1}^{M} y_c \ln(p_c), \tag{6}$$

where in general, y_c is the actual probability of the token, but in our case we consider this to be 1 when examining the correct token, and 0 when examining every other token. M is the total amount of tokens in the model vocabulary, and p_c is the probability that the model attributed to that token. After we have calculated the loss, we can backpropagate the loss through the network like discussed in Section 3.2.4 to get the best adjustment to all the weights and biases of the model, on average for the batch we considered.

4 KetGPT: Transformers for QASM Code Generation

In this thesis KetGPT is presented: KetGPT is a transformer model that is trained to generate realistic-looking QASM code. Generating QASM code in this way has its own specific challenges like: the training dataset is small compared to datasets normally used in machine learning, due to the fact that there are few known quantum algorithms, or that code, in general, is sensitive to syntax errors. In this section, applications for these circuits are elaborated, more practical matters are discussed and the experimental method is described.

4.1 Applications

The main thing differentiating circuits generated by KetGPT from circuits generated by the qiskit random_circuit implementation, is how "realistic" they are. Because they are closer to actual quantum circuits, they are more fit for certain quantum circuit mapping applications. In this section some of the main applications will be mentioned.

4.1.1 Benchmarking

The first reason why we would want to generate quantum circuits is for benchmarking purposes. When benchmarking different compilers for example, we would like to compare their performance on a large set of realistic quantum circuits. As mentioned in Section 2.4, some benchmark suits [19,23] include random circuits in benchmarks because of the lack of real quantum algorithms.

Example experiments could be, but are not limited to: comparing gate depth between different compiler optimisation passes on the same circuit, comparing gate depth between different compiler mapping passes on the same circuit, comparing gate depth or execution time on different compiler scheduler passes and comparing metrics between different full-compiler passes altogether.

4.1.2 Machine Learning

Promising applications within the field of machine learning are training a Reinforcement Learning compiler [7,8] using KetGPT circuits instead of random circuits, like suggested in [8], and in a similar way enhancing the deep neural network approach from [24] with KetGPT circuits that more closely resemble realistic circuits than the random circuits that were used in their approach. Intuitively, it makes sense to train a compiler to be good at handling realistic circuits rather than training it to be good at handling random circuits. Furthermore, many possibilities could be unlocked because machine learning usually requires large datasets to succeed, which now might become available due to KetGPT.

4.1.3 More Fundamental Quantum Information Research

Generating circuits that look more like real, useful, quantum circuits also opens up the question: “what does it mean for a quantum circuit to be useful?”. Analysing KetGPT circuits could help in answering this question, for example by analysing the average amount of entanglement that is present at certain phases in the circuit. This could help us think about what quantum hardware implementations would be best suited to run quantum circuits, even for algorithms that have not yet been invented.

4.2 Dataset Definition

4.2.1 Raw Dataset

Some collections of quantum algorithm datasets that can be used for benchmarking exist [19–21,23], for this thesis MQT Bench [19] was used. QASM files were created that describe circuits implementing algorithms for 2 to 100 qubits using OPEN-QASM 2.0 [6]. In some cases algorithms were not compatible with a certain amount of qubits (for example if an algorithm requires an uneven amount of qubits), for these algorithms all valid circuits within this range were created. The full dataset and more information on this dataset is supplied in Section 9.2.

4.2.2 Dataset Preprocessing

To get the dataset in a suitable format for the transformer models, we need to do a preprocessing step:

Minor adjustments to the circuits from the dataset were made to make the dataset more structured (for example consistently using a newline after every valid QASM statement), and because of technical limitations, large files cannot be processed by the model, so the maximum length of the circuits (measured in amount of valid QASM statements) that were used was 1024. Circuits that exceeded this length were not taken into account. This technical limitation is related to RAM limitations of the hardware that was used, and is not a general technical limitation. After the preprocessing step, the final dataset consists of 713 QASM files.

4.3 Generator

4.3.1 Architecture

For text and code generation, it is common to use a decoder-only transformer architecture [25]. Therefore, to generate the QASM files, the GPT-2 model architecture [40], which uses a decoder-only transformer has been chosen. Python code constructing this architecture is openly available using the GPT-2 implementation in the Huggingface “Transformers” python library [44, 45]. This architecture is a decoder-only transformer as is described in Figure 6, but where the layer normalisation was moved to the input of each sub-block instead of the output, and an additional layer normalisation was added after the final self-attention block. Furthermore, the way the weights are initialised when defining the model was changed to account for the effect of the amount of residual connections [40].

4.3.2 Tokenisation

As discussed in Section 3.2.1, a tokenisation scheme will be used to convert the dataset text into tokens. The original implementation of GPT-2 uses a certain type of tokenisation called Byte Pair Encoding (BPE). An intuitive way of understanding this method of tokenisation is that it divides up text in components (for example “training” can be split up into “train” and “ing” which makes it easier to capture the meaning of the full word “training”). A problem with using this method of tokenisation is that it makes it possible to generate QASM code that is not syntactically correct (for example the line “hh q0q1;” could theoretically be generated using this tokenisation method).

That is why, for the generator, the tokenisation method was changed to only allow syntactically correct QASM code as tokens. This was done by adjusting the GPT2Tokeniser class. A list of all valid QASM statements in the dataset is compiled, and that list is used as our vocabulary. This way it can be guaranteed that whatever token is generated by the model will be a valid QASM statement.

4.4 Classifier

After the QASM files have been generated using the generator, one wants to know if the generated files are indeed “realistic”. A binary classifier was used to classify whether a generated QASM file is more similar to files from our dataset, or it is more similar to a randomly generated file.

4.4.1 Architecture

For the classifier, an encoder-only transformer model as described in Section 3.3 was used. More specifically, the exact architecture that was used is the architecture of the DistilBERT model [46], using the implementation from the Huggingface transformers library [45]. This model is a smaller version of the highly influential encoder-only BERT model [41]. A smaller model in general ensures faster training and inference.

4.4.2 Tokenisation

For the generator it was necessary to create a custom tokenisation method to ensure that the generator could only generate valid QASM code. For the classifier, it is not necessary to use a custom tokenisation method, since the classifier will not generate code, but is only used for classification tasks, so the tokenisation method that was used to train the original DistilBERT model can be used. This tokenisation method is called WordPiece [47]. Similar to the BPE tokeniser briefly described in Section 4.3.2, it divides words into sub-words. The way these sub-words are chosen is what differentiates WordPiece from BPE, but is not relevant for this thesis.

The tokeniser truncates the QASM sequences after 512 tokens (since these tokens are now sub-words instead of QASM lines, 512 tokens coresponds to about 50 lines of QASM code, depending on the sequence) to make the files compatible with the maximum input size of the classifier model that was used.

4.5 Experimental Method

In this subsection some practicalities regarding producing the results of this thesis will be further specified.

4.5.1 Google Colab

To run the code that produces the results of this thesis, a Jupyter notebook [48] was executed in the Google Colab environment [49]. This Notebook is provided in Section 9.1. Google Colab allows users to access a powerful Tesla T4 GPU for free. The Tesla T4 has 16Gb of GDDR6 memory, 320 Turing tensor cores and 2560 CUDA cores, which is useful since a powerful GPU can significantly speed up machine learning model training and inference [50].

4.5.2 Python and Relevant Packages

Google Colab currently uses Python version 3.10.12. Relevant packages for the code used to obtain the results of this thesis are the transformers [44] (version 4.34.0) and datasets [51] (version 2.14.5) libraries from Huggingface, PyTorch [52] (version 2.0.1+cu118) and NumPy [53] (version 1.23.5).

4.5.3 Generator Model and Training Settings

Firstly, in Table 2 the parameters that define the structure of our generator model are specified. Default values correspond to the values used in the original GPT-2 implementation [40].

n_embd is the dimension of the word embedding vector (which was called *dim* in Section 3.2.2).

n_layer determines how many decoder blocks the model consists of. This value was set to 3 instead of the default of 12 to gives us a more lightweight model that is easier to train, uses up less memory and generates faster once the model is trained, at negligible cost of performance.

n_head determines the amount of attention heads in a multi-head attention block. This value was set to 4 instead of the default of 12, for the same reasons as for *n_layer*.

n_positions, which determines the maximum sequence length the model is able to process, was left at the default value of 1024.

vocab_size specifies the amount of tokens in the vocabulary. This is important because the amount of tokens in the vocabulary also determines the size of the output layer and, therefore has architectural implications. In this model the vocabulary size was set to 48291, since this corresponds to the total amount of unique QASM lines in our dataset.

Table 2: Generator model settings

Name	**Value**
n_embd	768 (default)
n_layer	3
n_head	4
n_positions	1024 (default)
vocab_size	48291

Table 3: Training settings

Name	**Value**
Epochs	5
Learning Rate	5e-5 (default)
Batch Size	4
Optimiser	AdamW (default)
Loss function	Cross-entropy (default)

The training settings are specified in Table 3.

Epochs determines how many times the model goes through the dataset during the training process. This parameter of course directly influences training time, but is mostly important because it also plays a large part in how well trained the model will be. If you don't train long enough (low amount of epochs), the model might not have been given enough time to adjust its weights to well represent the training data. But if you train for too long, the model might develop tunnel vision: representing the training data really well, but not being able to generalise further than the training data (this phenomenon is called overfitting [54]).

Learning rate is the α in equation (2) from section 3.2.4 which determines how large the update of the weights and biases of the model will be per step. Setting this parameter too low might result in very slow learning, but setting this parameter too high makes it possible for the optimisation algorithm to overshoot the optimal value for the weights (since the adjustments to the model are not fine enough).

Batch size is the amount of data points that are used in one optimisation step. The optimiser updates the weights that on average improve the model the most. Choosing a higher batch size makes the model train faster, but is more memory intensive.

The *Optimiser* is the algorithm that determines how the weights are updated at every step. AdamW [55] is similar to the algorithm that was described in Section 3.2.4, but is slightly more advanced.

The *loss function* quantifies how well our model performs. Cross-entropy is the standard loss function for classification problems (like predicting which word should be next, out of a vocabulary of words) [41].

4.5.4 Classifier Model and Training Settings

In Table 4 the settings that were used to define the classifier model are specified.

Table 4: Classifier model settings

Name	Value
n_embd	768 (default)
n_layer	6 (default)
n_head	12 (default)
n_positions	512 (default)
vocab_size	30522

Table 5: Training settings

Name	Value
Epochs	3
Learning Rate	5e-5 (default)
Batch Size	4
Optimiser	AdamW (default)
Loss function	Cross-entropy (default)

These settings have the same meaning as corresponding settings in the generator, except for the *vocab_size*: in the generator model the *vocab_size* is the dimension of the output layer, but in the classifier, the output layer dimension is 2 because we are doing binary classification instead of next token prediction. In the classifier, the *vocab_size* parameter is therefore more closely linked to the classifier tokeniser than to the classifier model itself as the parameter does not have architectural implications.

4.6 Generation Details

The generator model workflow is as follows:

4.6.1 Preparation

Generating tokens using the generator model is done as follows: i) a list is made that contains the qubit count for every algorithm in the dataset and a list is made that contains the amount of gates for every algorithm in the dataset; ii) random qubit count and a number of gates are chosen from those lists, and that will be the qubit count and amount of gates for the QASM file that will be created; and finally, iii) using these parameters, all invalid QASM statements related to qubit count will be filtered out (e.g., if the qubit count is 5, then all gates that involve qubit 13 won't be considered). This is done by making it impossible for the generator model to generate these tokens.

4.6.2 Model Input

The model will receive as input the following:

```
OPENQASM 2.0;

include "qelib1.inc"

qreg q[{}];
```

where {} will contain the chosen qubit count. This is the way all the QASM files in our dataset start, and it gives us an opportunity to control the qubit count in a simple manner.

4.6.3 Generation Scheme

Every time a new probability distribution over the tokens is generated, the 5 most probable tokens are selected, and from them, a new token is chosen according to the renormalised probability distribution over those 5 tokens (the probability is renormalised to ensure that all probabilities add up to 1). This gives us a way to introduce more randomness in the QASM file generation, while also keeping the generated tokens realistic since the 5 most probable tokens are probably still good candidates. This generation scheme is called top-k generation [56]. For this thesis, 5 was chosen as the relevant parameter, but in general, the amount of most probable tokens that are selected is a free parameter of the top-k generation scheme.

Furthermore, it is specified that it should not be possible that somewhere in the file a sequence of 15 tokens repeats itself. Although this is not ideal for QASM code generation (since algorithms often contain repeating sequences), in transformer model generation, it sometimes happens that the model gets stuck in a loop, predicting the same sequence over and over again, and this is to be prevented.

Lastly, this process is simply repeated until the desired amount of gates is reached.

4.7 Classification Details

When training the classifier, a dataset is set up where all real quantum algorithms in the MQTbench dataset are labeled “0” (1112 QASM files), and an equal amount of QASM files, that consists of gates that are randomly chosen from a list of all unique QASM statements in the dataset, are labeled “1”. To make sure that the classification process is as fair as possible, similar to how the KetGPT generated QASM files were generated, it is made sure that the randomly generated QASM files will contain the same distribution of qubit counts and amount of gates as the original dataset.

Since the classifier model only accepts up to 512 tokens, the dataset is tokenised in a way such that the maximum amount of tokens is 512, which corresponds to about 50 QASM lines (depending on the QASM file). All lines after that are truncated. The downside of this approach is that in determining if the QASM file is real or not, the full QASM file is not taken into account, but only the first part of it, but the advantage is that training and inference is much faster and a less technically complex model is needed. Furthermore, one can usually already tell from the first part of a QASM file whether it is random or if it has some logical structure.

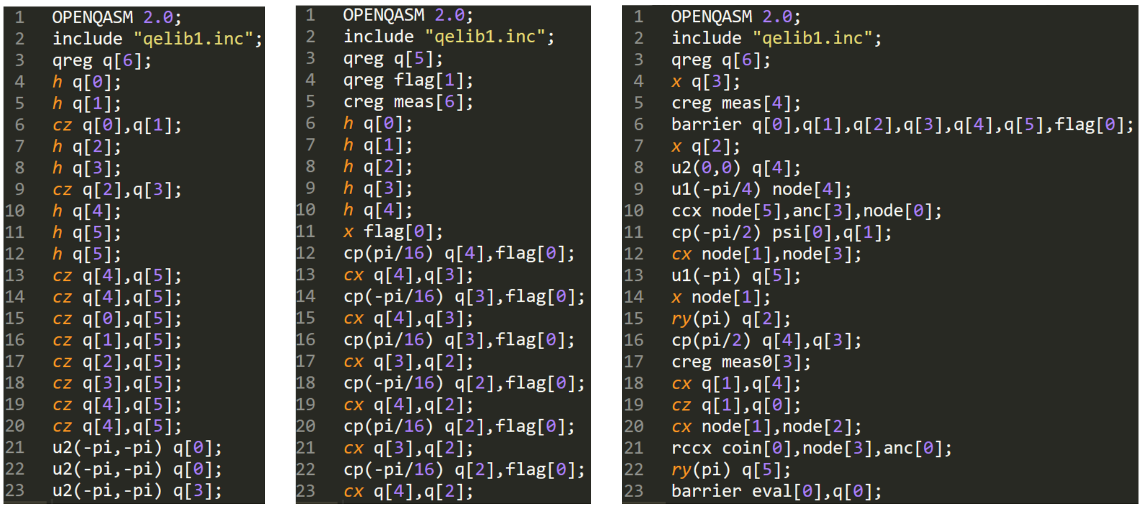
Then the model is trained on the labeled dataset, and afterwards the trained model is used to predict whether the KetGPT generated circuits are labeled “0” or “1”, meaning that they are closer to actual algorithms or random algorithms respectively.

5 Results

In this section the results of the thesis will be presented. Although KetGPT is still a proof of concept, so the results can likely be drastically improved and should therefore be regarded as the results of a minimal working version, the results demonstrate that the approach for generating realistic-looking quantum circuits used in this thesis is promising.

5.1 Visual Inspection

First we visually inspect (the first lines) of a circuit generated by KetGPT, and compare it to what the first lines of a real and a fully random circuit look like:



```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[6];
h q[0];
h q[1];
cz q[0],q[1];
h q[2];
h q[3];
cz q[2],q[3];
h q[4];
h q[5];
h q[5];
cz q[4],q[5];
cz q[4],q[5];
cz q[0],q[5];
cz q[1],q[5];
cz q[2],q[5];
cz q[3],q[5];
cz q[4],q[5];
cz q[4],q[5];
u2(-pi,-pi) q[0];
u2(-pi,-pi) q[0];
u2(-pi,-pi) q[3];
```

(a) KetGPT

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
qreg flag[1];
creg meas[6];
h q[0];
h q[1];
h q[2];
h q[3];
h q[4];
x flag[0];
cp(pi/16) q[4],flag[0];
cx q[4],q[3];
cp(-pi/16) q[3],flag[0];
cx q[4],q[3];
cp(pi/16) q[3],flag[0];
cx q[3],q[2];
cp(-pi/16) q[2],flag[0];
cx q[4],q[2];
cp(pi/16) q[2],flag[0];
cx q[3],q[2];
cp(-pi/16) q[2],flag[0];
cx q[4],q[2];
```

(b) Real

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[6];
x q[3];
creg meas[4];
barrier q[0],q[1],q[2],q[3],q[4],q[5],flag[0];
x q[2];
u2(0,0) q[4];
u1(-pi/4) node[4];
ccx node[5],anc[3],node[0];
cp(-pi/2) psi[0],q[1];
cx node[1],node[3];
u1(-pi) q[5];
x node[1];
ry(pi) q[2];
cp(pi/2) q[4],q[3];
creg meas0[3];
cx q[1],q[4];
cz q[1],q[0];
cx node[1],node[2];
rccx coin[0],node[3],anc[0];
ry(pi) q[5];
barrier eval[0],q[0];
```

(c) Random

Figure 7: Side by side comparison between the first lines of a 6 qubit QASM file generated by KetGPT, taken from a real dataset and a random circuit.

Note that the lines in the KetGPT file and in the real file contain structure, like repetition of hadamard and 2-qubit gates (cx and cz), whereas the fully random circuit does not contain such a repetitive sequence. Also note that the order in which the hadamard gates is applied in the KetGPT and the real circuit is ascending in qubit number, whereas in the fully random circuit, as is to be expected, there is no logical order of operations. It is important to mention that the random circuit contains invalid statements (for example an operation on node 4 is instructed, but node 4 was never defined), but this error is also sometimes present in files generated by KetGPT, although, seemingly, less often. The fact that it is not specifically forbidden for KetGPT to generate invalid statements, but it still generates such statements considerably less often than random files, can also be seen as a realistic feature of KetGPT generated data.

Some more examples of circuits generated by KetGPT are presented as supplementary information in Section 9.3.1.

From these examples, and from the example in Figure 7, it can be concluded that visual inspection strongly suggests that KetGPT circuits contain features of real quantum circuits, demonstrating that the approach of using transformers to generate quantum circuit data is promising.

5.2 Classifier Model evaluation

In an attempt to get a quantifiable measure, a classifier model was trained to classify a QASM file as real, or as randomly generated. This was done using training data that consisted of real quantum circuits, and of random quantum circuits. The classifier was trained using this training data, but to evaluate its performance, it is important to use data that it has not seen in training before (so-called test data). A subset of 15% of the total data, that is has not yet encountered during the training process, is fed to the classifier model, and used to evaluate the performance of the classifier. To evaluate this performance, a confusion matrix can be used to determine how the predictions of the model relate to the actual labels of the data. In Figure 8, the confusion matrix that corresponds to this test evaluation is presented:

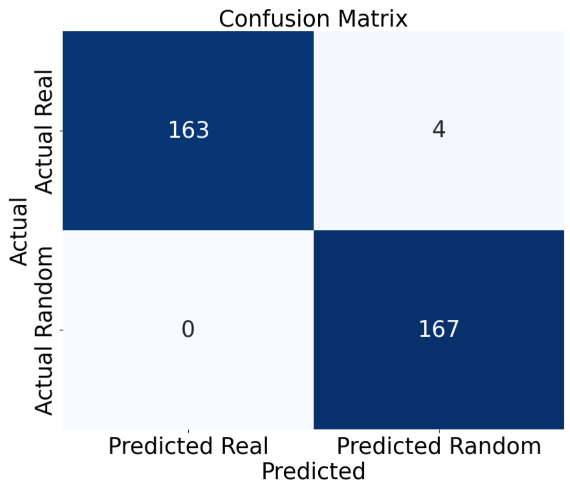


Figure 8: Classifier performance on a test dataset illustrated by a confusion matrix. Diagonal values are correctly predicted, only 4 QASM files that were actually “True” were predicted as being “Random”

The total test dataset consisted of 334 (167 real and 167 random) entries, and 330 of the dataset values were predicted correctly, that means that the classifier model achieved an accuracy of 98.8%. Which would mean that the classifier model is capable of discerning real quantum circuits from random quantum circuits in 98.8% of the cases.

Then, instead of predicting on real algorithms and completely random algorithms, the classifier model was asked whether 1000 QASM files that were generated with KetGPT are closer to the algorithms from the MQTBench dataset (the real algorithms), or whether they are closer to completely random algorithms. 997 out of 1000 circuits were classified as being real, which is 99.7%.

The classifier model accuracy however is suspiciously high, and after visual inspection of the circuits, it seemed that there was a strong correlation between whether measurement operations were present in the circuit and the circuit being classified as random. This makes sense since in realistic circuits, a measurement operation is much more likely to be present towards the end of the circuit, as opposed to in the first ~50 lines of code (remember, the classifier can only see the first part of the QASM code), but completely random circuits don't have this property. On the one hand, the classifier captures an important feature of realistic QASM code, namely that it is unrealistic to have measurement operations so early in a circuit. But on the other hand, it is not desirable for the classifier to be strongly influenced by only a single feature, because other features like sequence repetition or other examples of logical structure might not receive the attention they deserve.

The process was repeated, but this time all real, random and KetGPT QASM files were rewritten such that there are no measurement operations present.

The resulting confusion matrix is presented in Figure 9.

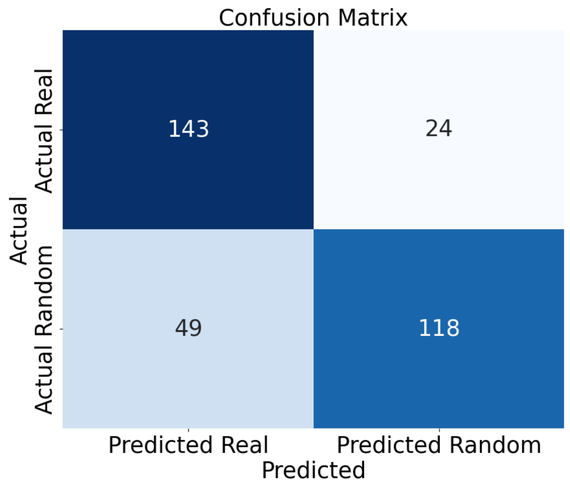


Figure 9: Confusion matrix for classifier trained on real and random data where all measurement operations were removed.

261 out of 334 files from the dataset were correctly classified, giving the model an accuracy of 78.1% on the test dataset. This accuracy is much lower than the accuracy of the previous model which was 98.8%, which could be explained by the fact that the problem it needs to solve is harder: the model can't simply check if there is a measurement operation and use that to determine whether the sequence is real or random anymore. After classifying these real and random circuits, KetGPT circuits were investigated. This classifier model classified 486 out of the total of 1000 circuits as being closer to the real algorithm dataset than to the random circuits, which is 48.6%. This value is quite low, especially since the confusion matrix in Figure 9 shows that the model more often classifies random circuits as real, than that it classifies real circuits as random. Since it is easy to discern by visual inspection which QASM files from the test dataset are real and which ones are random, an accuracy of 78.1% might be an indication that the model is underfitting, i.e. the model was not being trained enough to be able to capture complex dependencies.

One final classifier model was trained for 6 epochs instead of 3, to investigate whether the previous model could be underfitting. The confusion matrix that corresponds to this model can be seen in Figure 10.

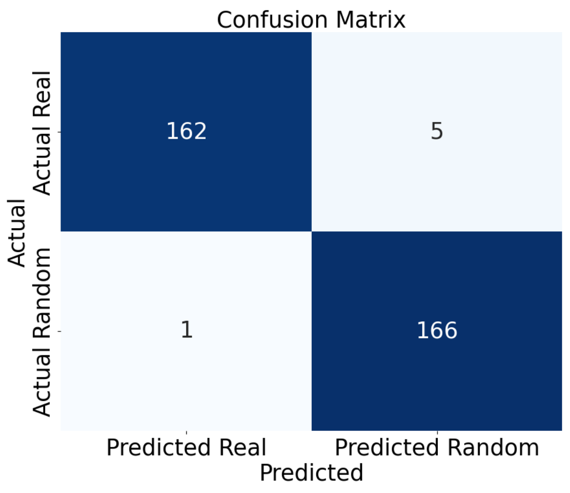


Figure 10: Confusion matrix for classifier trained for 6 epochs, on real and random data where all measurement operations were removed.

This confusion matrix is more similar to the confusion matrix of the original classifier model from Figure 8, where the measure operations were still present in the dataset. This time 328 out of 334 QASM files from the test dataset were predicted correctly, which gives the model an accuracy of 98.2%. Moreover, this classifier model classifies 999 out of the 1000 KetGPT circuits as real, which is 99.9%.

It is difficult to determine how reliable each model is: as mentioned earlier, the problem the classifier has to solve is easy, as we can also easily tell by visual inspection which QASM files are random and which ones are written by a human, which could explain the extremely high accuracy of some of the classifier models. The high accuracy can also be explained by the fact that the test dataset consists, for lack of a better alternative, of a random subset of the total data. It is therefore impossible to have, for example, the Deutsch–Jozsa algorithm [57] on 6 qubits in both the training dataset, and the test dataset. But it **is** possible that Deutsch-Jozsa on 6 qubits is in the training dataset, and Deutsch-Jozsa on 5 qubits is in the test dataset. Since they are very similar, the classifier model is now being tested on (semi) training data, and then the accuracy is not a reliable metric anymore. This phenomenon is called data leakage.

The random QASM files in the test set however, are not necessarily similar to the random files in the training dataset, and are predicted correctly every time. This could be an indication that the classifier model result is reliable after all.

Taking these considerations into account, none of the classifier models seem to be reliable enough to use as a definitive classifier. Since the first classifier seemed to pick up realistic features in the data (the measurement operations being present), looking at its accuracy on the test dataset, it is at least plausible that the final classifier model also picks up some feature(s) of realistic QASM files. But whether this means that it also overfits on a feature, or it has learned relevant aspects of realistic QASM files is difficult to tell.

In conclusion, the proposed classifier models are not able to solve the problem of quantifying how realistic-looking a quantum circuit is. Nevertheless, visual inspection of KetGPT files still provides good reason to believe that these KetGPT circuits contain features of real quantum circuits.

5.3 Training and Inference Time

All computations were done using the hardware described in Section 4.5.1. KetGPT training time was 240 seconds, and generating 1000 QASM files took 8818 seconds (147 minutes), which means that on average KetGPT needs 8.8 seconds per generated file. However, the QASM files are of varying size (as explained in Section 4.6.1), and the amount of time needed to generate one file is non-linearly dependent on the size of the file, so this number should be taken as a rough estimate of generation time per file.

6 Discussion

6.1 Sequentiality of Qubits

A human way of writing QASM code is by starting with performing e.g. a gate on qubit 0, and then performing a gate qubit 1. But in principle the qubit number is just a name and doing the same operation on qubit 43 and qubit 22 is equivalent to performing the operations on qubit 0 and 1. KetGPT consistently generates files in the way humans would write them (with qubit numbers in ascending order) because it was trained on a dataset of QASM code that was written by humans. On the one hand this is desirable, since it makes sense to for example benchmark a compiler with code that looks like it was written by a human, because that's what it should be good at. But on the other hand it might be that KetGPT pays too much attention to the qubit number, where the gate it performs is more important in terms of implementing a realistic quantum algorithm.

6.2 Variability of Generated QASM Files

To determine if a collection of QASM files is useful, they need to be realistic individually, but this criterion does not suffice: if all of the generated files are similar, or even the same, then as a whole they are not useful, even though they could be very realistic. By visual inspection it seems like there is some repetition of common patterns in the generated files. See for example KetGPT 20 and KetGPT 793 in Section 9.3.1: the QASM files both build up the entanglement in the qubits in a similar way using hadamards and cz gates. More examples of repeating patterns are found in the full dataset. Some similarity however is to be expected, since the dataset also consists of QASM files that have similarities, even if they describe different quantum algorithms. Furthermore, there are only so many realistic ways to build up entanglement in your qubits, and most of them involve hadamard gates, so some repetition is to be expected. Ultimately, it seems as though there is a reasonable amount of variability in the generated files, but as of now, this variability was not quantified in a reliable way and is left for future work.

6.3 Test Data Leakage

As mentioned in Section 5.2, the test dataset for the classifier model contains data leakage: the test data contains information about the training data because the QASM files for the same algorithm using different qubit counts are similar. Since the amount of real quantum algorithms is so limited, building a proper validation dataset is also difficult. One possible solution would be to train the classifier on the MQTBench dataset, like was done for this thesis, and to test the classifier on QASM files describing the same algorithms, but in a different QASM description, by using a different dataset source for example. But this approach has major downsides as well since the classifier is being trained on defining features of QASM files from one source, and these defining features might not be able to translate well to QASM files from other sources. Furthermore, data leakage is still present since the same algorithms will be used for training and testing, just from a different source.

Another option would be to not train the classifier on one algorithm, and use QASM files that implement that algorithm as a test dataset. But then the test dataset essentially only consists of one datapoint, in which case the sample size is too small. Even if we train many different classifiers, each time leaving out a different algorithm, the classifiers are all different so if they all individually perform well on their own respective test datasets, it is still not possible to confidently say that a classifier works well. Therefore, the methodology of testing the classifier used in this thesis was chosen, but visual inspection should be the most important measure, which should only be aided by the classifier model.

6.4 Random Circuit Depth and Qubit Count

KetGPT and random circuits were generated according to the method described in Section 4.6.1. Quickly summarised, the amount of qubits and the amount of QASM statements are chosen randomly from their corresponding distribution in the real algorithms dataset. The amount of QASM statements, and the amount of qubits that are used in a specific QASM file are chosen individually, while in real circuits the qubit count correlates with the amount of gates. This methodology of choosing the amount of qubits and the amount of QASM statements to generate QASM files was chosen so that in training on the random circuits, the classifier might learn that long QASM files with low qubit count are more likely to be synthetic, which is a property which is desirable for the classifier to have. It has not yet been investigated if the classifier indeed learned such a skill.

6.5 Consistency of Generation Quality

Only the first lines of QASM files were investigated, which means that any conclusion that can be drawn from this investigation does not have to hold for other parts of the QASM files. It could be the case that after a certain number of generated statements, the quality of generated statements decreases. This effect has not been extensively investigated, and it is therefore not possible to conclude that the QASM files in their entirety are realistic, although visual inspection does not indicate a clear decrease in quality.

6.6 Useful Quantum Algorithms

The circuits generated by KetGPT are not meant to be interpreted as circuits that implement useful quantum algorithms. The circuits are realistic-looking and might describe some undiscovered quantum algorithms, but without knowing what the algorithm should be doing, it is near impossible to reverse engineer what the algorithm a circuit is implementing is useful for.

6.7 Notable Limitation

Furthermore, because this version of KetGPT is merely a proof of concept, the circuits generated by KetGPT are still discernible from real quantum circuits, and it is challenging to do fundamental research about what properties quantum algorithms tend to have, like suggested in Section 4.1.3.

7 Conclusion

Transformer models, which are used in realistic text and code generation, have caused a paradigm shift in the field of natural language processing, and simultaneously, quantum computers are being developed which are promising since they could solve certain problems faster than classical computers can. However, datasets of real quantum circuits, which are crucial for benchmarking these quantum computers and which are crucial for various machine learning applications in the field of quantum circuit mapping, are scarce.

Therefore the problem statement for this thesis was:

"Can transformer models be used to generate realistic-looking quantum circuits to expand quantum algorithm datasets?"

The main contribution presented in this thesis was KetGPT, a decoder-only transformer model that is able to generate realistic-looking quantum circuits. Circuits that were generated by KetGPT were presented, which, by visual inspection, are easily distinguishable from random circuits, and show desirable qualities like structure and humanlike programming factors like applying gates in the order of ascending qubits.

Furthermore, classifier models were suggested in an attempt to get a quantifiable measure to determine if a given QASM file is likely to be realistic, or whether it was likely to be randomly generated. The final classifier model reached an accuracy of 98.2% on the test dataset, correctly predicting 328 out of 334 QASM files, and classified 999 out of the 1000 KetGPT generated circuits as real. However, due to data leakage, it is unclear if the classifier behaves as intended. But although it is not possible to use the trained classifier model to unambiguously quantify how realistic the QASM files generated by KetGPT are, the results from the classifier model do make it plausible that there is an amount of realism to these QASM files, as the classifier model did seem to be able to pick up features of real quantum circuits.

In conclusion, the quality of the files can not reliably be quantified by the classifier model, but visual inspection strongly suggests that KetGPT quantum circuits contain features of real quantum circuits, demonstrating that the novel approach of using transformers to generate data for quantum circuit mapping purposes is promising.

8 Future Work

8.1 Hyperparameter tuning

Both KetGPT and the classifier model have not undergone any hyperparameter tuning. Although it is difficult to do so without a reliable validation dataset, for which there are not enough algorithms, improvements can be made by adjusting hyperparameters like, but not limited to: the amount of epochs the models are trained for, the learning rate, the embedding vector size, the amount of layers, the amount of heads and many more parameters. This hyperparameter tuning process is difficult to perform methodically however, as it is not possible to create a validation dataset without it suffering from data leakage.

8.2 Other Generation Methods

As mentioned in Section 4.6.3, top-k generation has been used to generate the QASM files that are presented in this thesis. There are some other generation schemes like top-p [58], beam search [59] and contrastive search [60], that are used in text generation tasks. In future work, different generation methods could be compared, and a generation scheme that is specifically apt for QASM file generation could be used instead of the aforementioned standard text generation schemes.

8.3 Discrete Token Set

Every unique QASM statement was converted into a token. That means that the token set is discrete. This method was chosen as it was the easiest implementation to make the proof of concept that KetGPT was supposed to be, but in general this is not satisfactory way to represent building blocks for general QASM files since some real QASM files contain arbitrary angles. To resolve this, an arbitrary gate token could be created, which in post-processing could be filled in by a transformer specifically trained for this purpose.

8.4 Adjusting Tokenisation Scheme

As mentioned in the previous section, as of now the QASM code is tokenised into full QASM statements where for example “hadamard gate on qubit 1” is one token. KetGPT might perform better if it is trained on data where the gates and the target qubits are seperated tokens, like “hadamard gate” being a token, and “on qubit 1” being another token. If the generation scheme is adjusted accordingly, it can still be guaranteed that only valid QASM expressions are generated. This approach also scales better into higher qubit counts.

9 Supplementary Information

9.1 Notebook

The code that was used for this thesis is provided as a Jupyter notebook [48], which was executed in the Google Colab environment [49], from the following url:

https://colab.research.google.com/drive/1dbtJX6q8sED4yrb1I09KUuXWYH0AVN8r?usp=sharing

9.2 Data

The data that was used in this thesis can be found from the following url:

https://www.kaggle.com/datasets/boranapak/ketgpt-data

It contains the dataset that was used, and a KetGPT folder that contains: the KetGPT model, the KetGPT tokeniser, the classifier model, all KetGPT generated circuits and all random circuits.

The 1112 real algorithm QASM files in the dataset, were generated using MQTbench [19] using the algorithms:

- Amplitude Estimation (AE)
- Deutsch-Jozsa
- Graph State
- GHZ State
- Grover's (no ancilla)
- Grover's (v-chain)
- Portfolio Optimization with QAOA
- Portfolio Optimization with VQE
- Quantum Approximation Optimization Algorithm (QAOA)
- Quantum Fourier Transformation (QFT)
- QFT Entangled
- Quantum Neural Network (QNN)
- Quantum Phase Estimation (QPE) exact
- Quantum Phase Estimation (QPE) inexact
- Quantum Walk (no ancilla)

- Quantum Walk (v-chain)
- Variational Quantum Eigensolver (VQE)
- W-State
- Ground State
- Pricing Call Option
- Pricing Put Option

And using the settings:

- qubit range: 2-100
- Target-independent level: Qiskit

9.3 QASM Files

9.3.1 Selection of KetGPT Generated Files

To give an impression of the files KetGPT generates, a random number generator was used to pick 5 numbers between 0 and 999. The first 30 lines of KetGPT files corresponding to those random numbers are presented.

KetGPT 20:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[34];
h q[0];
h q[1];
h q[3];
cz q[0],q[1];
h q[2];
h q[3];
h q[3];
h q[4];
h q[5];
h q[6];
h q[7];
cz q[6],q[7];
h q[8];
h q[9];
cz q[8],q[9];
h q[10];
h q[11];
h q[12];
```

```
cz q[11],q[12];
h q[13];
h q[14];
h q[15];
h q[15];
h q[16];
h q[17];
h q[18];
h q[19];
```

KetGPT 398:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[42];
h q[1];
u2(0,0) q[0];
u2(0,0) q[1];
h q[2];
u2(0,0) q[3];
h q[4];
u2(0,0) q[5];
u2(0,0) q[6];
h q[7];
u2(0,0) q[8];
u2(0,0) q[9];
h q[10];
u2(0,0) q[11];
u2(0,0) q[12];
h q[13];
h q[14];
u2(0,0) q[15];
h q[16];
h q[17];
h q[18];
h q[19];
h q[20];
u2(0,0) q[21];
h q[22];
h q[23];
u2(0,0) q[24];
u2(0,0) q[25];
```

KetGPT 409:

```
OPENQASM 2.0;
```

```
include "qelib1.inc";
qreg q[97];
creg meas[4];
ry(-pi/4) q[0];
ry(-pi/3) q[2];
cz q[0],q[1];
h q[2];
h q[3];
cz q[2],q[3];
h q[4];
cz q[3],q[4];
h q[5];
measure q[0] -> meas[0];
measure q[1] -> meas[1];
measure q[2] -> meas[2];
measure q[3] -> meas[3];
measure q[4] -> meas[4];
measure q[5] -> meas[5];
measure q[6] -> meas[6];
measure q[7] -> meas[7];
measure q[8] -> meas[8];
measure q[9] -> meas[9];
measure q[10] -> meas[10];
measure q[11] -> meas[11];
measure q[12] -> meas[12];
measure q[13] -> meas[13];
measure q[14] -> meas[14];
measure q[15] -> meas[15];
measure q[16] -> meas[16];
```

KetGPT 793:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[14];
h q[0];
h q[0];
h q[1];
cz q[0],q[1];
h q[2];
h q[3];
h q[4];
h q[5];
h q[6];
h q[7];
h q[8];
```

```
h q[9];
h q[10];
h q[11];
h q[12];
h q[13];
cz q[12],q[13];
h q[12];
h q[13];
cz q[12],q[13];
h q[13];
cz q[12],q[13];
cz q[12],q[13];
cz q[12],q[13];
cp(-pi/8) q[3],q[0];
cp(pi/2) q[12],q[11];
h q[11];
```

KetGPT 807:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[31];
h q[1];
ry(-pi/4) q[0];
ry(-pi/3) q[2];
h q[2];
h q[3];
h q[4];
u2(0,0) q[5];
h q[5];
h q[6];
h q[7];
h q[8];
h q[9];
h q[10];
h q[11];
h q[12];
h q[13];
h q[14];
h q[15];
h q[16];
h q[17];
cz q[16],q[17];
h q[18];
h q[19];
h q[20];
```

```
h q[21];
h q[22];
h q[23];
```

9.3.2 Real QASM Files Classified as Random

These QASM files were from the real dataset, but were classified as random **Quantum Walk (no ancilla) 3 qubits:**

```
OPENQASM 2.0;
include "qelib1.inc";
qreg node[2];
qreg coin[1];
creg meas[3];
h coin[0];
ccx coin[0],node[1],node[0];
cx coin[0],node[1];
x coin[0];
x node[1];
ccx coin[0],node[1],node[0];
cx coin[0],node[1];
u2(-pi,-pi) coin[0];
x node[1];
ccx coin[0],node[1],node[0];
cx coin[0],node[1];
x coin[0];
x node[1];
ccx coin[0],node[1],node[0];
cx coin[0],node[1];
u2(-pi,-pi) coin[0];
x node[1];
ccx coin[0],node[1],node[0];
cx coin[0],node[1];
x coin[0];
x node[1];
ccx coin[0],node[1],node[0];
cx coin[0],node[1];
x coin[0];
x node[1];
barrier node[0],node[1],coin[0];
```

Quantum Walk (no ancilla) 7 qubits:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg node[6];
qreg coin[1];
creg meas[7];
h node[0];
cu1(pi/32) node[5],node[0];
cx node[5],node[4];
cu1(-pi/32) node[4],node[0];
cx node[5],node[4];
cu1(pi/32) node[4],node[0];
cx node[4],node[3];
cu1(-pi/32) node[3],node[0];
cx node[5],node[3];
cu1(pi/32) node[3],node[0];
cx node[4],node[3];
cu1(-pi/32) node[3],node[0];
cx node[5],node[3];
cu1(pi/32) node[3],node[0];
cx node[3],node[2];
cu1(-pi/32) node[2],node[0];
cx node[5],node[2];
cu1(pi/32) node[2],node[0];
cx node[4],node[2];
cu1(-pi/32) node[2],node[0];
cx node[5],node[2];
cu1(pi/32) node[2],node[0];
cx node[3],node[2];
cu1(-pi/32) node[2],node[0];
```

Quantum Walk (no ancilla) 8 qubits:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg node[7];
qreg coin[1];
creg meas[8];
h node[0];
cu1(pi/64) node[6],node[0];
cx node[6],node[5];
cu1(-pi/64) node[5],node[0];
cx node[6],node[5];
cu1(pi/64) node[5],node[0];
cx node[5],node[4];
cu1(-pi/64) node[4],node[0];
cx node[6],node[4];
cu1(pi/64) node[4],node[0];
cx node[5],node[4];
cu1(-pi/64) node[4],node[0];
```

```
cx node[6],node[4];
cu1(pi/64) node[4],node[0];
cx node[4],node[3];
cu1(-pi/64) node[3],node[0];
cx node[6],node[3];
cu1(pi/64) node[3],node[0];
cx node[5],node[3];
cu1(-pi/64) node[3],node[0];
cx node[6],node[3];
cu1(pi/64) node[3],node[0];
cx node[4],node[3];
cu1(-pi/64) node[3],node[0];
```

Quantum Walk (no ancilla) 12 qubits:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg node[11];
qreg coin[1];
creg meas[12];
h node[0];
cu1(pi/1024) node[10],node[0];
cx node[10],node[9];
cu1(-pi/1024) node[9],node[0];
cx node[10],node[9];
cu1(pi/1024) node[9],node[0];
cx node[9],node[8];
cu1(-pi/1024) node[8],node[0];
cx node[10],node[8];
cu1(pi/1024) node[8],node[0];
cx node[9],node[8];
cu1(-pi/1024) node[8],node[0];
cx node[10],node[8];
cu1(pi/1024) node[8],node[0];
cx node[8],node[7];
cu1(-pi/1024) node[7],node[0];
cx node[10],node[7];
cu1(pi/1024) node[7],node[0];
cx node[9],node[7];
cu1(-pi/1024) node[7],node[0];
cx node[10],node[7];
cu1(pi/1024) node[7],node[0];
cx node[8],node[7];
cu1(-pi/1024) node[7],node[0];
```

Quantum Walk (v-chain) 15 qubits:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg node[8];
qreg coin[1];
qreg anc[6];
creg meas[15];
h coin[0];
rccx coin[0],node[1],anc[0];
rccx node[2],anc[0],anc[1];
rccx node[3],anc[1],anc[2];
rccx node[4],anc[2],anc[3];
rccx node[5],anc[3],anc[4];
rccx node[6],anc[4],anc[5];
ccx node[7],anc[5],node[0];
rccx node[6],anc[4],anc[5];
rccx node[5],anc[3],anc[4];
rccx node[4],anc[2],anc[3];
rccx node[3],anc[1],anc[2];
rccx node[2],anc[0],anc[1];
rccx coin[0],node[1],anc[0];
rccx coin[0],node[2],anc[0];
rccx node[3],anc[0],anc[1];
rccx node[4],anc[1],anc[2];
rccx node[5],anc[2],anc[3];
rccx node[6],anc[3],anc[4];
ccx node[7],anc[4],node[1];
x node[1];
rccx node[6],anc[3],anc[4];
```

9.3.3 Random QASM Files Classified as Real

This QASM file was randomly generated, but was still classified as real by the final classifier

```
OPENQASM 2. 0;
include "qelib1.inc";
qreg q[63];
cp(0) eval[1],eval[48];
cz q[11],q[44];
cx q[20],q[36];
cx q[9],q[51];
cp(0) q[53],q[9];
cx q[0],q[45];
cz q[22],q[28];
cx q[21],q[25];
```

```
u1(-pi) q[6];
h eval[19];
```

9.3.4 KetGPT Files Classified as Random

The only KetGPT file that was classified as random per the final classifier **KetGPT 463:**

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[78];
ry(-pi/4) q[0];
ry(-pi/4) q[0];
ry(-pi/3) q[2];
h q[14];
rccx coin[0],node[1],anc[0];
rccx coin[0],node[1],anc[0];
rccx coin[0],node[2],anc[0];
ry(-pi/3) q[2];
rccx coin[0],node[1],anc[0];
rccx node[2],anc[0],anc[1];
rccx coin[0],node[2],anc[0];
x node[1];
rccx coin[0],node[1],anc[0];
rccx coin[0],node[2],anc[0];
rccx node[3],anc[0],anc[1];
rccx coin[0],node[2],anc[0];
rccx node[3],anc[0],anc[1];
rccx coin[0],node[2],anc[0];
rccx node[3],anc[0],anc[1];
x node[1];
rccx node[3],anc[0],anc[1];
h q[20];
rccx node[2],anc[0],anc[1];
rccx coin[0],node[1],anc[0];
rccx coin[0],node[2],anc[0];
x node[1];
```

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] OpenAI. Chatgpt. https://openai.com/chatgpt. Accessed: 15-11-2023.

[3] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[5] Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2(1):1–8, 2016.

[6] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[7] Stan van der Linde, Willem de Kok, Tariq Bontekoe, and Sebastian Feld. qgym: A gym for training and benchmarking rl-based quantum compilation. *arXiv preprint arXiv:2308.02536*, 2023.

[8] Thomas Fösel, Murphy Yuezhen Niu, Florian Marquardt, and Li Li. Quantum circuit optimization with deep reinforcement learning. *arXiv preprint arXiv:2103.07585*, 2021.

[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[10] Stephen Jordan. Quantum algorithm zoo. https://quantumalgorithmzoo.org/. Accessed: 25-09-2023.

[11] Troels F Rønnow, Zhihui Wang, Joshua Job, Sergio Boixo, Sergei V Isakov, David Wecker, John M Martinis, Daniel A Lidar, and Matthias Troyer. Defining and detecting quantum speedup. *science*, 345(6195):420–424, 2014.

[12] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[13] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[14] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Phys. Rev. Lett.*, 103:150502, Oct 2009.

[15] Aritra Sarkar. Automated quantum software engineering: why? what? how? *arXiv preprint arXiv:2212.00619*, 2022.

[16] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[17] Adriano Barenco. A universal two-bit gate for quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 449(1937):679–683, 1995.

[18] Medina Bandic, Hossein Zarein, Eduard Alarcon, and Carmen G Almudever. On structured design space exploration for mapping of quantum algorithms. In *2020 XXXV conference on design of circuits and integrated systems (DCIS)*, pages 1–6. IEEE, 2020.

[19] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. Mqt bench: Benchmarking software and design automation tools for quantum computing. *Quantum*, 7:1062, 2023. MQTbench is available at https://www.cda.cit.tum.de/mqtbench/.

[20] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM Transactions on Quantum Computing*, 4(2):1–26, 2023.

[21] Robert Wille, Daniel Große, Lisa Teuber, Gerhard W Dueck, and Rolf Drechsler. Revlib: An online resource for reversible functions and reversible circuits. In *38th International Symposium on Multiple Valued Logic (ismvl 2008)*, pages 220–225. IEEE, 2008.

[22] Teague Tomesh, Pranav Gokhale, Victory Omole, Gokul Subramanian Ravi, Kaitlin N Smith, Joshua Viszlai, Xin-Chuan Wu, Nikos Hardavellas, Margaret R Martonosi, and Frederic T Chong. Supermarq: A scalable quantum benchmark suite. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 587–603. IEEE, 2022.

[23] Medina Bandic, Carmen G Almudever, and Sebastian Feld. Interaction graph-based characterization of quantum benchmarks for improving quantum circuit mapping techniques. *Quantum Machine Intelligence*, 5(2):40, 2023.

[24] Giovanni Acampora and Roberto Schiattarella. Deep neural networks for quantum circuit mapping. *Neural Computing and Applications*, 33(20):13723–13743, 2021.

[25] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[26] Andrea Agostinelli, Timo I Denk, Zalán Borsos, Jesse Engel, Mauro Verzetti, Antoine Caillon, Qingqing Huang, Aren Jansen, Adam Roberts, Marco Tagliasacchi, et al. Musiclm: Generating music from text. *arXiv preprint arXiv:2301.11325*, 2023.

[27] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[28] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning internal representations by error propagation, 1985.

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[30] Jay Alammar. The illustrated transformer, 2018. `http://jalammar.github.io/illustrated-transformer/` Accessed: 5-10-2023.

[31] Enrico Briano, Claudia Caballini, Pietro Giribone, Roberto Revetria, et al. Neural network models for the management of tests in power plants. In *Proceedings of the 9th WSEAS international conference on System science and simulation in engineering*, pages 319–326. World Scientific and Engineering Academy and Society (WSEAS), 2010.

[32] John Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. *Advances in neural information processing systems*, 2, 1989.

[33] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[34] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[35] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: The basic theory. In *Backpropagation*, pages 1–34. Psychology Press, 2013.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[37] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. arXiv preprint arXiv:1607.06450, 2016.

[38] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[39] Chelsy Ma. The intuition behind context extension mechanisms for llms, 2023. https://medium.com/@machangsha/the-intuition-behind-context-extension-mechanisms-for-llms-b9aa036304d7 Accessed: 27-10-2023.

[40] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[41] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[42] Jiahuan Lei, Qing Zhang, Jinshan Wang, and Hengliang Luo. Bert based hierarchical sequence classification for context-aware microblog sentiment analysis. In *Neural Information Processing: 26th International Conference, ICONIP 2019, Sydney, NSW, Australia, December 12–15, 2019, Proceedings, Part III 26*, pages 376–386. Springer, 2019.

[43] Irving John Good. Rational decisions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 14(1):107–114, 1952.

[44] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. pages 38–45. Association for Computational Linguistics, October 2020.

[45] Hugging Face Inc. Transformers: State-of-the-art natural language processing. https://github.com/huggingface/transformers, 2021. Accessed: 13-10-2023.

[46] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108, 2019.

[47] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[48] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. *Elpub*, 2016:87–90, 2016.

[49] Google LLC. Google colaboratory, 2023. Retrieved from https://colab.research.google.com.

[50] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880, 2009.

[51] Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, Joe Davison, Mario Šaško, Gunjan Chhablani, Bhavitvya Malik, Simon Brandeis, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Clément Delangue, Théo Matussière, Lysandre Debut, Stas Bekman, Pierric Cistac, Thibault Goehringer, Victor Mustar, François Lagunas, Alexander Rush, and Thomas Wolf. Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[53] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[54] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 2002.

[55] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[56] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[57] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

[58] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

[59] Alex Graves. Sequence transduction with recurrent neural networks. *arXiv preprint arXiv:1211.3711*, 2012.

[60] Yixuan Su, Tian Lan, Yan Wang, Dani Yogatama, Lingpeng Kong, and Nigel Collier. A contrastive framework for neural text generation. *Advances in Neural Information Processing Systems*, 35:21548–21561, 2022.