Universiteit Leiden

NQS-Bench-101: Benchmarking Neural Architecture Search for Neural Quantum States

Piotr Lichota (s4072006)

Supervisors:
Dr. Anna Dawid-Lekowska & Bartosz Kreft

Bachelor Thesis– Data Science and Artificial Intelligence

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

August 3, 2026

Abstract

Neural quantum states (NQS) are a general class of variational quantum states parameterized in terms of an artificial neural network. NQS typically approximate the amplitude of the quantum wavefunction using a limited number of computational resources due to the exponential size of the Hilbert space. Although we can approximate any continuous function using a neural network with a certain structure, it remains unclear which MLP architectural choices improve NQS performance. This study systematically trains and evaluates thousands of NQS architectures to discover insights about architectural choices of the search space and create a tabular benchmark for reproducible evaluation of NAS algorithms. The transverse field Ising model (TFIM) features spins with nearest-neighbor interactions as well as an external magnetic field. The regression task solved by NQS in this setting is to predict a ground state amplitude given a spin configuration. The goal of NQS is to predict this ground-state amplitude as accurately as possible, minimizing regression losses. Neural architecture search (NAS) is the process of automating the design of neural architectures for a given task. Inspired by NAS research about benchmarking NAS algorithms and techniques, this study explores the nature of the search space of a large number of NQS architectures. This exploration derives insights into which groups of architectures positively influence learning the quantum wavefunction. This study demonstrates that neural architecture choices strongly influence wave-function amplitude prediction performance, while increasing computational resources alone does not necessarily guarantee improved results. Carefully selected architectures can achieve high predictive performance while maintaining low complexity. NAS experiments further show that evolutionary search provides stable optimization, although random search remains a competitive alternative.

Contents

1 Thesis overview **1**

2 Introduction to machine learning **1**
2.1 Machine learning and cognitive inspiration . . . 1
2.2 Learning process in deep learning . . . 1
2.3 AI fundamental building blocks . . . 2
2.4 Limitations . . . 2
2.5 The rise of computational power . . . 2
2.6 Tasks . . . 2
2.7 Learning approaches . . . 3
2.8 Artificial neural networks . . . 3
2.9 Universal approximation theorem . . . 4
2.10 Neural network architecture . . . 4
2.11 Neural network training . . . 5
2.11.1 Training in supervised learning . . . 5
2.11.2 Training, validation and test sets . . . 5
2.11.3 Backpropagation algorithm and gradient descent . . . 6
2.11.4 Epochs, batches and gradient updates . . . 6
2.12 Automated machine learning . . . 6

3 Neural architecture search **7**
3.1 Traditional NAS . . . 7
3.2 Neural architecture search components . . . 8
3.2.1 NAS as a subfield of AutoML . . . 8
3.2.2 NAS pipeline . . . 9
3.2.3 Search space . . . 9
3.2.4 Search strategy . . . 10
3.2.5 Algorithms overview . . . 10
3.2.6 Different training pipelines . . . 11
3.2.7 Speedup techniques . . . 12
3.2.8 NAS and hyperparameter optimization . . . 12
3.3 NAS goals . . . 12
3.4 Applications . . . 13
3.5 Task definition . . . 13

4 Background **13**
4.1 The transverse field Ising model . . . 13
4.2 The quantum many-body wavefunction . . . 14
4.3 Neural quantum states . . . 15
4.3.1 Parameterizing quantum wavefunction . . . 15
4.3.2 Double descent . . . 15
4.3.3 Spin configurations . . . 16
4.3.4 Dataset composition . . . 16

4.3.5 Loss functions . . . 16

5 Related Work **17**
5.1 Benchmarking . . . 17
5.2 NAS-Bench-101 . . . 17
5.2.1 Motivation . . . 17
5.2.2 Search space . . . 17
5.2.3 Role in research . . . 18
5.2.4 Algorithms analyzed . . . 18
5.3 Other benchmarks and applications . . . 18
5.3.1 NAS-Bench-201 . . . 19
5.3.2 NAS-Bench-301 . . . 19
5.3.3 NAS-Bench-Suite . . . 19
5.4 Research gap . . . 19

6 Approach **20**
6.1 Research questions . . . 20
6.2 Datasets . . . 20
6.2.1 Explanation and reasoning why particular h values were chosen . . . 21
6.2.2 Amplitude distribution . . . 21
6.3 Search space . . . 22
6.3.1 Overall network structure . . . 22
6.3.2 Activation functions . . . 22
6.3.3 Network widths and depths . . . 23
6.4 Evaluation metrics . . . 23

7 Experiments **24**
7.1 Hyperparameter optimization . . . 24
7.2 Training . . . 24
7.2.1 ALICE . . . 25
7.2.2 Scheduler . . . 25
7.2.3 Results Storage and Organization . . . 26

8 Results **26**
8.1 Result comparison over different regimes . . . 26
8.2 Architecture dataset characteristics . . . 30
8.2.1 Full architecture dataset . . . 30
8.2.2 Activation functions vs. performance . . . 32
8.2.3 Network depth vs. performance . . . 34
8.2.4 Network width vs. performance . . . 36
8.2.5 Training time vs. performance . . . 37
8.2.6 Trainable parameters vs. performance . . . 40
8.3 NAS algorithms . . . 41
8.3.1 Evolutionary search . . . 42

9 Conclusions and Further Research **43**

References **49**

1 Thesis overview

The 2 section contains the basic necessary knowledge from the fields of artificial intelligence, deep learning, the universal approximation theorem, and machine learning. The 3 section contains information and a description of neural architecture search, including its components, goals, and applications. Section 4 introduces how machine learning can be used to solve a well-known quantum physics problem. The 5 section contains prior work on neural architecture search and its role in research. The 6 section explains the research questions, datasets used to solve the machine learning task, evaluation metrics, and an extensive description of the search space used in this study. Section 7 contains information on the experiments conducted to answer the research questions. Section 8 discusses the statistical results obtained and their interpretations. Section 9 contains the conclusions, interpretations based on the obtained results, and ideas for future research.

2 Introduction to machine learning

Machine learning (ML) is a subfield of artificial intelligence that focuses on creating algorithms that learn decision logic from data [46]. The mechanism is not explicitly programmed with the steps the computer must follow [13], [1]. An example of such a technique is an automatic credit approval system for a bank. Machine learning learns patterns from examples instead of relying on strict rules created by a programmer (e.g. if-else statements).

By learning from data, the ML algorithms identify patterns and use them to make predictions. The difference can be described with a simple analogy: baking a cake. Classical programming is similar to following a recipe for a cake, where it is exactly known what to do and what the outcome will be [54]. In contrast, machine learning learns from observations, much like learning from a master baker who bakes a cake. The results might vary depending on the subtleties picked up [54].

2.1 Machine learning and cognitive inspiration

The machine learning representation learning process is inspired by human perception and representation [6]. Intelligent behavior in humans can be partly described by recognizing one concept through multiple modalities [8]. By way of illustration, we can create a mental image of a cat. Its image can be created by drawing, painting, or seeing it [13].

2.2 Learning process in deep learning

The domain of deep learning uses highly parameterized models to derive complex patterns from data, achieving state-of-the-art performance [36]. In the supervised learning context, the machine learning model is designed to solve a specific task using data and corresponding target labels (e.g. image classification). To perform this task, an ML algorithm, when provided with data, learns to identify patterns. During the training process, the model's parameters are adjusted based on the provided data to improve performance metrics [21] (see Section 6.4). These metrics quantify prediction error relative to the ground truth in the data [13]. Training an accurate model typically requires sufficient computational resources and training time.

2.3 AI fundamental building blocks

The main goal of an AI system is to make machines perform intelligent tasks similar to those performed by humans [29, 13, 45]. Achieving such a goal of intelligent behavior requires three fundamental building blocks. The first component is knowledge representation, which focuses on how machines store and organize real-world information [42]. The second ingredient is learning, which relates to how the system learns patterns or models. The learning process happens based on interaction with data and identifying patterns. The third component is called reasoning, where the system makes decisions and predictions based on its accumulated knowledge and newly collected data.

2.4 Limitations

When we implement knowledge representation, learning, and reasoning in the computer world, these capabilities are associated with machines that can "reason" and solve problems. The question of whether machines can truly think is a highly debatable topic [55]. Nevertheless, machines solve many well-defined computational tasks by applying formal rules and mathematical methods within a computational framework. In such domains, machines outperform humans, as demonstrated in games such as chess and Go [33, 28, 9, 52]. In contrast, humans still outperform machines in tasks that are difficult to formulate mathematically, such as understanding intentions, interpreting social contexts, and moral reasoning.

2.5 The rise of computational power

The main cause of the success of deep learning in recent years was the increased availability of computational resources, parallel algorithms, and training datasets [34]. Due to these advances, deep neural network training became more common and computationally feasible [13]. This enabled the development of larger machine learning models and solving more complex tasks. The deep learning approach allows for the analysis of large-scale datasets for prediction tasks and identifying patterns that humans cannot detect given the amount of data involved. Examples of applications include face recognition or disease classification from images. Due to the increasing availability of AI technologies in society, AI is now a daily tool used by many people, as it is able to solve problems with high predictive performance in many real-world domains.

2.6 Tasks

One of the typical machine learning tasks is regression, where in a basic setting we assume a linear relationship between two variables x and y (input and output target) [22]. Typically, x represents one or more input features, and y represents the target value. The objective in this setting is to find a function that fits all tuples (x, y) in the training data and generalizes well to unseen instances. However, in many regression models, the relation is non-linear (the input and output data cannot be represented as a straight line). In the regression machine learning task, we can use a numerical variable (such as income) to predict a credit score for a particular person in an automatic credit issuing system.

Another very popular machine learning task is classification, where we try to assign a discrete class to an input instance, sample, or observation. There are three basic kinds of classification

tasks. In the simplest settings, classification is binary, when we have only two categories (e.g., predicting whether an email is spam or not). However, there are problems requiring multiple-class classification, where a single data instance has more than two possible classes and each input instance belongs to exactly one of those classes. An example of such a multiclass classification task is predicting a handwritten digit from an image. In this setting, an image can be assigned to only one of the ten labels indicating the digit that has been drawn in this image. The third kind of classification task is multi-label classification, where a single training instance can hold more than one label. An example of this task is the identification of multiple objects in a single image.

2.7 Learning approaches

In supervised learning, the data in an ML system can be understood as a set with several instances, which can be accompanied by a label or target for each feature data entry (e.g., a single entry can be a single row where the last column is the target label). Each column corresponds to a single feature, which can be of varied data types.

The type of dataset and knowledge we have affects the choice of learning paradigm in AI. In machine learning, there are three basic types of learning approaches: supervised, unsupervised, and reinforcement learning. In supervised learning, we have immediate access to the ground truth labels we want to optimize our model for. In the supervised learning setup, algorithms learn from labeled data where there is a clear objective to optimize. However, creating a model with robust performance requires correctly labeled data, which is not always available. In reinforcement learning, the agent learns through trial and error in order to maximize cumulative rewards [53]. This learning task is called reinforcement learning, where the agent interacts with the environment and receives feedback based on its actions [17]. In the reinforcement learning setting, the training data are generated by interacting with the environment, rather than providing it beforehand. There is an environment with which to interact to gain experience on how to act based on rewards and the exploration–exploitation trade-off [2]. In this dilemma, the algorithm must choose either to explore new paths or to utilize its already learned knowledge about the environment to accomplish its goal. Based on these rewards, the agent gains new experience over multiple steps. The unsupervised learning approach has to gain new knowledge from the provided data in order to discover structure, representations, or patterns. One of the unsupervised machine learning general goals is identifying the structure and patterns that exist in the data (without labels). In the unsupervised ML setting without corresponding labels, similar data points are grouped, which helps to identify hidden patterns in data.

2.8 Artificial neural networks

In recent years, the dominant choice for realizing the three learning approaches has been an artificial neural network (ANN). An ANN is a computational model inspired by biological neural networks and neurons present in neural circuits of living organisms [16, 38]. The ANN consists of interconnected nodes (artificial neurons) with adjustable weights. The artificial neurons are typically organized in layers where the data flows through each layer of this network [3]. The artificial neurons consist of compositions of a nonlinear activation function and a linear function (see Figure 1). Without non-linearity of activation functions, even the most sophisticated neural architectures would collapse into simple linear models with severe limitations. The entire network

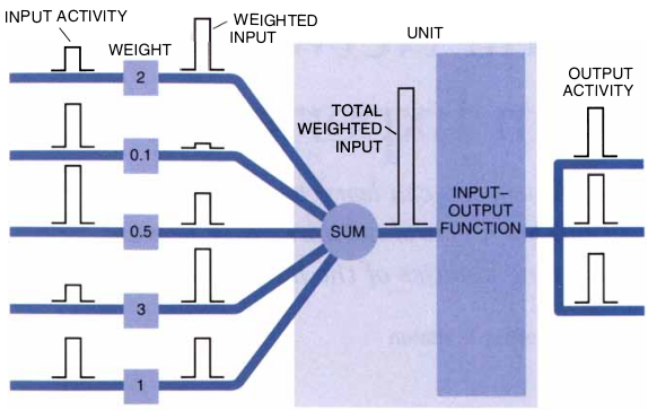


Figure 1: A visualization of information flow in an artificial neuron. Each input activity is multiplied by a number called a weight. Those inputs are then added together. The resulting output is calculated by an input-output (activation) function. Taken from [25]

reduces to a single linear transformation. Non-linearity enables complex pattern recognition and expands the representational capacity of neural networks [35]. The simplest form of artificial neural networks is feedforward neural network (FNNs) [43]. It consists of an input layer, hidden layers, and an output layer. ANNs can model complicated relationships in the data.

2.9 Universal approximation theorem

The universal approximation theorem (UAT) states that such neural networks with at least one hidden layer can approximate arbitrary continuous functions within an arbitrary degree of accuracy [3].

The theory states that there exists a sufficiently large network (with a limited number of neurons) that can approximate arbitrary functions with a desired accuracy. In this case, a continuous function is represented as a superposition of simpler components (also known as basis functions). The major results and theorems on the approximation capabilities of NNs state that depth-2 NNs with a suitable activation function are universal approximators. NNs of depth 1 have limited approximation capability [3].

While UAT states that such a network exists, it says nothing about the topology of those networks; how to build such an architecture that gives an acceptable approximation [12, 27]. The architecture design step involves human intervention. A neural architecture designer must decide about architecture configuration such as network depth, width in each (hidden) layer, and operations such as activation functions.

2.10 Neural network architecture

When L indicates the number of hidden layers, the total number of layers in a network is $L + 2$ (where we have L hidden layers, an input and output layer). Each layer of this neural network can be represented as:

$$\mathbf{y}_i = \mathbf{f}_i(\mathbf{x}_i) = \sigma_i(\mathbf{A}_i\mathbf{x}_i + \mathbf{b}_i). \tag{1}$$

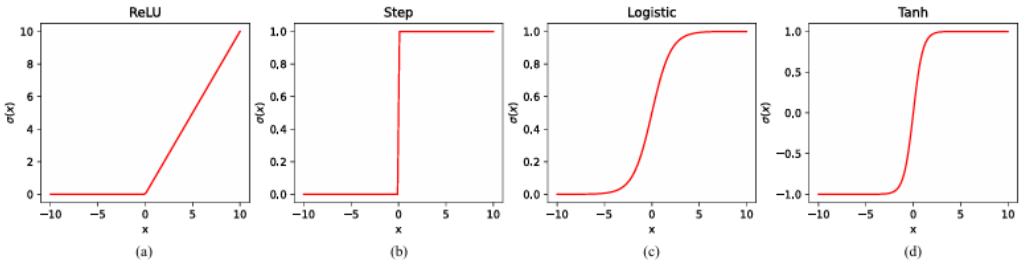


Figure 2: Overview of the most popular activation functions. Taken from [3].

Where $\mathbf{x}i$ indicates the input to the ith layer and $\mathbf{y}_i$ denotes the output of the i_{th} layer. The activation function in the i_{th} layer is denoted by σ_i. The matrix of learnable weights, in the neural network, is denoted by A_i. The bias vector is indicated by $\mathbf{b}_i$. The most popular non-linear activation functions are $ReLU$, $Tanh$, $Logistic$, and $Step$ [41] (see Figure 2).

In the input layer, there is only one neuron per feature, and it is simply passed as $\sigma_0(\mathbf{A}_0\mathbf{x} + \mathbf{b}_0)$. The neural network with only one hidden layer is called a shallow neural network. A deep neural network is a neural network with more than one hidden layer.

2.11 Neural network training

2.11.1 Training in supervised learning

The training of a neural network is based on updating the weights between the input, hidden units, and output. First, the neural network is presented with training examples. In supervised learning, each training example has data to give to the network, as well as the correct output—the desired answer the neural network should produce. After the neural network prediction, it is determined whether the network input matches the desired output [25]. Based on the network prediction and the desired response, an error is calculated, where a typical measure for regression tasks is mean squared error (MSE). Based on the computed loss, an optimization algorithm and gradients, the weights of each connection are updated to produce a better approximation of the desired output. The training process consists of multiple iterations over the training data, denoted as epochs (for details, see Sections 2.11.3 and 2.11.4).

2.11.2 Training, validation and test sets

The dataset used for evaluating a machine learning model is divided into training, validation, and test sets [7]. This split enables model training, hyperparameter tuning, and unbiased model evaluation. The training set is used to fit a certain ML algorithm to find the model parameters. The validation set is used to evaluate different model configurations and tune the corresponding hyperparameters. The test set serves as a final evaluation and the test set is independent: it is held out from feature selection, training, optimization, and validation. If the performance of the model on the test set is not satisfactory, the model needs to be further refined [37], [25]. This refinement should be based on the training and validation sets. One common example used for training, validation, and test split in machine learning is 70%, 20%, and 10%, respectively.

2.11.3 Backpropagation algorithm and gradient descent

The backpropagation algorithm is one of the most important methods for training neural networks. The algorithm optimizes weights and biases by computing the gradients of the loss function with respect to the network parameters [44]. The name of the algorithm originates from propagating the error backward through the network and utilizing chain rule to calculate the contribution of each parameter to the final error. The backpropagation algorithm pipeline consists of two main steps: forward and backward pass. In the forward pass, the data is propagated layer-by-layer. In each layer, a linear transformation and an activation function is applied to the input of this layer. The output of one layer becomes an input to the next layer. The difference between the output of the last layer of the network (final prediction) and the target value (ground truth) is quantified using a predefined loss function (e.g. MSE loss). In the backward pass step, the chain rule is applied to compute the gradients of the loss function with respect to the network parameters. The backward process is repeated layer-by-layer from output to input, where contribution of each weight and bias to the total error is determined. The gradient descent algorithms are then used to update model's learnable parameters based on the computed gradients. In gradient descent algorithms, each weight is adjusted based on the learning rate which determines the step size during optimization [44, 23, 10].

2.11.4 Epochs, batches and gradient updates

In a single epoch, the model sees the entire training set, which is grouped into batches to utilize parallel processing. Instead of loading the entire dataset into memory at once, the data set is split into smaller batches. The total loss in each epoch is calculated per batch by summing up all prediction errors (calculated by the loss function) for each data point.

During the training process, gradients of the loss are computed with respect to the model parameters (see Section 2.11.3). The run of the *train* function involves running three intermediate steps. The first step clears the gradients of all optimized tensors. It avoids gradient accumulation, ensures correct gradient calculation, and prevents memory leaks.

The second step calculates the slope at the current position by determining how each parameter contributes to the error and what the optimal direction is to move toward reducing the error. The gradients are computed based on the slope calculated in the backward pass. The third step updates the model parameters using the optimizer and the gradients computed in the previous step.

2.12 Automated machine learning

The rapid expansion of big data, high-performance computing, and tremendous demand for hands-free machine learning solutions gave rise to the field of automated machine learning (AutoML) [32, 30]. Although machine learning has achieved considerable success in recent years and a growing number of disciplines rely on it, the ML success relies on human machine learning experts who preprocess the data, construct appropriate features, model, design neural networks, and analyze the results obtained [32]. In the traditional ML pipeline, human engineers need to select the right neural architectures, training procedures, regularization methods, and hyperparameters. The complexity of those tasks is sometimes beyond non-ML experts.

The AutoML domain helps to make machine learning pipelines accessible without expert knowledge. It allows exploration more possible alternatives for solutions, reducing the time for

manual design and trial-and-error [5].

Automated machine learning provides methods to improve the efficiency of machine learning and accelerate research on ML [32]. AutoML targets the progressive automation and "democratization" of machine learning, where the ML domain is accessible to a wider range of people [30, 18]. The current goal of AutoML research is to semi-automate those pipeline steps and gradually remove the human in the loop [30].

One emerging subfield of AutoML is called neural architecture search (NAS), which targets the problem of finding well-performing deep neural networks [60]. Neural architecture search is one of the 3 ingredients along with hyperparameter optimization and meta-learning [15] (see Section 3). It has been shown that AutoML methods are mature enough and are able to surpass human machine learning experts in several tasks [30], [5]. The time required to perform trial and error on those tasks creates a significant amount of work that these experts need to spend to find optimal settings.

In contrast, the AutoML pipeline aims to automatically determine the optimal approach for this task based on data provided by the user. While the goal of AutoML is to minimize human intervention in solving machine learning tasks and save time, it still requires a basic level of understanding of machine learning algorithms and programming. While the automatic pipeline aims to discover optimal solutions, it can lead to lower model quality than manual training and opaque model complexity [4].

Nowadays, NAS is mistakenly equated with AutoML [30]. However, NAS is just one of the many ingredients of AutoML, and there is much more to AutoML than NAS. NAS is considered one of the most challenging tasks in AutoML, due to the extremely large search space and long training times of architectures.

3 Neural architecture search

The main objective for neural architecture search is to find a well-performing architecture for a given machine learning task. This architecture topology design is difficult, as the search space grows exponentially with an increasing number of architectural choices and the exhaustive search becomes infeasible. NAS has become an important research field, as it automates this process of finding an optimal architecture.

3.1 Traditional NAS

Traditional NAS relies on human-designed search spaces [56]. Architecture designers needed to carefully make architectural choices in order to discover an optimal architecture. However, the expertise of those people is limited and makes the system unusable for non-experts. The scope of the NAS domain has been limited to only a few tasks.

The huge choice of possible architecture designs boosted the development of NAS, which treats the discovery of a high-performing architecture for a given task as an optimization problem. NAS has produced architectures with state-of-the-art performance for various tasks, but image classification is the dominant domain in this field.

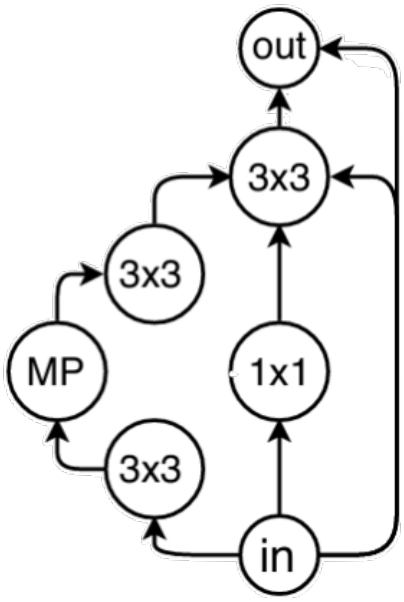


Figure 3: Illustration of a NAS-Bench-101 cell that attained the lowest mean test error. The nodes indicate operations, and the edges indicate how information flows between operations. A cell is represented as a directed acyclic graph (DAG).

3.2 Neural architecture search components

3.2.1 NAS as a subfield of AutoML

One of the key factors in the success of deep learning is the reduction in manual feature engineering, which requires a tremendous amount of computational resources, expertise, and time. In recent years, the emergence of high-performance deep learning architectures has resulted in successful breakthroughs in a variety of fields such as computer vision, natural language processing, and reinforcement learning [56]. Neural architecture search (NAS) is a subfield of machine learning in which the process of designing a well-performing architecture is automated. Given a machine learning task (such as image classification), NAS aims to discover high-performing architectures within a search space (see Section 3.2.3). Those architectures differ in the number of layers, parameters and operations. Even though NAS reduces the human effort for manual architecture design, it outperforms many hand-designed architectures on various tasks (such as classification on ImageNet) [56]. NAS focuses on automating the design of neural network architectures. Other components of the machine learning pipeline (such as hyperparameter optimization or data preprocessing) are handled by broader AutoML methods. NAS is often considered a closely related field to hyperparameter optimization (HPO), and can be viewed as a special case of HPO where architectural choices are included in the search space [57, 56]. Architecture optimization in NAS is much more complex because the dependence of previous design choices may affect the results of subsequent ones. The topology of an architecture is typically represented by a directed acyclic graph (DAG), where nodes represent operations, and edges represent the information flow between them. The network topology is defined by the arrangement of nodes and the connections (edges) between them (see Section 3.2.3).

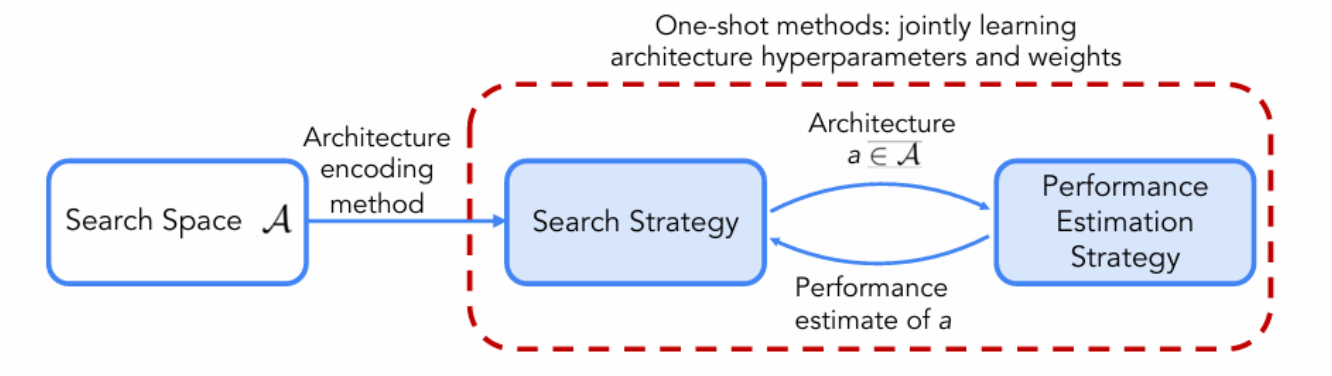


Figure 4: An overview of NAS pipeline, taken from [56]

3.2.2 NAS pipeline

Many modern NAS techniques incorporate weight sharing for similar architectures to avoid computationally expensive training. The weights of this supernet are shared among all candidate architectures [59]. This super-architecture is structured to cover the search space of candidate architectures. In classical NAS frameworks (as described in prior literature), a search strategy proposes architectures, and performance estimation evaluates them. The search strategy is updated based on the evaluation results [56]. Since one-shot methods reuse weights and train one large architecture, the search and performance estimation are tightly coupled within a single training process through shared weights (see Figure 4). An example of a performance estimation strategy is validation accuracy computed using shared weights. The one-shot approach reduces computational cost, but introduces bias in performance estimation due to weight sharing.

3.2.3 Search space

The search space in NAS research specifies all architectures the NAS algorithm is allowed to select and defines the set of candidate architectures that can be explored by the NAS algorithm (see Sections 3.2.3, 3.2.5 and 8.3). The search space is task-specific and usually cell-based. However, there are other types of search spaces that have also gained significant attention. Other types of search spaces include chain-structured and hierarchical search spaces. The atomic unit in building neural network topology typically consists of operations such as convolutions, pooling, normalization, or activation functions [56].

The cell-based search spaces boil down to searching over small cells (micro-architecture) instead of searching for the entire architecture (macro-architecture) [56]. The two most popular cell-based search spaces are DARTS (operation choices on the edges) and NAS-Bench-101 (operation choices on the nodes; see Figure 3 for illustration). In DARTS (Differentiable ARchitecTure Search), gradient descent and NAS are combined to automatically optimize neural network architectures in ANNs [24]. Cell-based search spaces are widely used in NAS due to their efficiency and transferability [56]. On the other hand, the choice of a cell-based search space reduces its expressiveness, making it difficult to discover truly novel architectures.

For a cell-based NAS search space, each cell can be represented as a directed acyclic graph with a maximum number of nodes, where each node holds a label indicating the corresponding operation for this node (cell-based search space grows exponentially with the number of nodes and possible

operations) [58].

The NAS search design represents a trade-off, where hand-picked decisions allow finding a high-performing architecture in less time, while a more extensive search space could allow the discovery of truly novel architectures that may have a more primitive design.

A typical NAS search space consists of layers where there is a choice of a few operations for each layer (such as 1x1 convolution or residual layer). Such units create a micro search space. A module indicates a sequence of those layers and creates a macro search space. Hierarchical search spaces allow for more than one searchable level [56].

3.2.4 Search strategy

Search strategies in NAS are one of the most studied NAS components and include black-box optimization methods (e.g., reinforcement learning, evolutionary algorithms) as well as weight-sharing (one-shot) approaches [56]. In black-box methods, each sampled architecture is trained independently to obtain performance estimates. In one-shot methods, a single over-parameterized model is trained, and shared weights are used to evaluate architectures. There are hybrid approaches that combine black-box and weight-sharing methods [56].

3.2.5 Algorithms overview

The comparison baseline for NAS algorithms is random search, where architectures are chosen randomly before the complete training process [56]. Another comparison baseline is local search, where neighbors of the best architectures found so far are chosen and differ by one operation or edge from the best architecture.

Popular NAS algorithms include evolutionary approaches where the population of architectures is iteratively updated [56]. Every step of an evolutionary algorithm consists of mutating "parent architectures", which perform well based on evaluation metrics, to create "child architectures". These new architectures are trained, and the population of architectures is updated by replacing the oldest architectures in the population or architectures with the worst performance. The regularized evolution algorithm drops the architecture that has lived the longest in the population, even if this longest-lived sample has the highest accuracy. This method achieved state-of-the-art performance on ImageNet.

In NAS reinforcement learning, there is sequential decision-making over architectures. The reward signal is typically based on validation performance. This performance is used to train a controller that learns to sample high-performing architectures [56]. However, reinforcement learning-based NAS has become less prominent in recent years compared to evolutionary approaches (for algorithm comparison, see Section 8.3).

The success of Bayesian Optimization comes from the efficient optimization of expensive functions. This method builds a probabilistic surrogate model and creates a function that balances exploration and exploitation during the search. The surrogate model is trained based on previously evaluated architectures. The architecture is chosen to maximize the acquisition function. At the end, the architecture with the highest validation accuracy is chosen [56].

Another class of NAS methods includes Monte Carlo Tree Search (MCTS) algorithms, which recursively sample new decisions by running stochastic rollouts to obtain a reward (yielding high-performing architectures) [56]. Then, the obtained reward is propagated back up the tree to update

node statistics such as visit counts and value estimates. This algorithm balances exploitation and exploration by performing decisions that build a tree search toward regions of the search space with high performance. To enable a higher number of MCTS rollouts, a neural network can be used. Further efficiency can be achieved by pruning the tree or applying multi-objective NAS.

Multi-objective NAS optimizes for multiple parameters. While many NAS settings optimize only a single metric (e.g., accuracy), multi-objective NAS solves a constrained optimization problem involving optimal latency, memory usage, and hardware constraints. The set of optimal solutions in this setting is called the Pareto front.

3.2.6 Different training pipelines

Another NAS concept is one-shot techniques, which target the immense computational cost of traditional three-ingredient NAS, where it is common to train each architecture independently from scratch (increasing the computational burden of those methods). In one-shot method, there is only a single training process of a hypernetwork (which generates weights conditioned on architecture encoding) or a supernetwork (which contains all possible architectures in the search space). Unlike supernets, which contain all candidate operations, hypernetworks generate the weights for candidate architectures using a separate neural network. Supernets enable parameter sharing across a large number of architectures, significantly reducing the cost compared to training each architecture independently due to the linear cost of adding new operations to the supernet. It has been debated whether performance under one-shot training is the same as when training each architecture independently.

Supernet training consumes a large amount of memory and latency and can only contain architectures with roughly the same size [56]. Treating architecture parameters as normal architecture weights may lead to improved generalization. In the hypernetwork approach, neural network weights are generated for multiple neural networks.

To save computational resources, other one-shot methods (such as DARTS) involve simultaneous training and search where discrete architecture choices are relaxed into continuous weights over operations. A black-box search strategy assumes that we have no access to any information about the objective function other than its inputs and outputs [20]. After training a supernet, a black-box search strategy can be used to select the best-performing sub-network based on shared-weight evaluation. Although they face performance issues and require a lot of expertise, they offer orders-of-magnitude speedup over black-box optimization techniques. The differentiable NAS methods allow the use of gradients to optimize their parameters. In differentiable supernet methods, gradient descent is used for a continuous relaxation to search for a high-performing local optimum. This method is used for DAG-based methods with different choices of operations on each edge. The DARTS method gained attention through simplicity and novelty, but the room for improvement led to a series of extensions [56]. DARTS has been reported to exhibit poor test generalization and to converge to sharp local minima in the loss landscape.

The weight inheritance method involves reusing weights from trained architectures to initialize similar candidate architectures, thereby accelerating convergence [56].

The meta-learning approach eliminates the need to train single architectures from scratch by leveraging previously explored solutions. Meta-learning-based NAS methods reuse knowledge from previously explored architectures or tasks to initialize or guide the search process.

3.2.7 Speedup techniques

The speedup techniques in NAS are divided into performance estimation, performance predictors, and multi-fidelity optimization. Besides training for fewer epochs, common speedup techniques include learning curve extrapolation, surrogate models, and zero-cost proxies [56].

The performance predictor in this context is defined as any function that predicts the accuracy of architectures without fully training them [56], which can save a significant amount of time.

Speedup techniques are used in NAS research to estimate the final validation performance of networks before they are fully trained, which can speed up the runtime of NAS algorithms. Their objective is to produce predictions that are highly correlated with the validation accuracy obtained after full training. Examples of such performance predictors include learning curve extrapolation, zero-cost proxies, partial learning curve models with parametric fitting, and surrogate models. Zero-cost proxies can be applied to Bayesian optimization NAS and one-shot NAS. Zero-cost proxies compute inexpensive statistics from a network using only one forward or backward pass, allowing architectures to be ranked without training. The drawback of zero-cost proxies is that they are sometimes unreliable and weakly correlated with fully trained performance.

Another class of speedup techniques is multi-fidelity optimization. Instead of training every architecture until convergence, these methods evaluate candidate architectures using cheaper approximations such as smaller datasets. The performance of such architectures is evaluated using cheaper, lower-fidelity versions of training. These methods introduce a fidelity parameter that determines how expensive the evaluation is. Higher fidelity corresponds to evaluations that more closely approximate full training. Examples of multi-fidelity algorithms include Successive Halving and HyperBand [56].

3.2.8 NAS and hyperparameter optimization

Hyper-parameters are configuration variables that control the behavior of machine learning algorithms. The choice of their values determines the effectiveness of systems based on these technologies. Manual hyperparameter search is often time consuming and becomes infeasible when the number of hyper-parameters is large. Automating the search is an important step towards advancing, streamlining, and systematizing machine learning [19]. The typical hyper-parameters include learning rate, optimizer, batch size or weight decay.

While hyperparameter optimization (HPO) and NAS are distinct optimization problems, HPO plays a significant role in NAS because it can lead to substantial performance improvements. NAS can be jointly combined with hyperparameter optimization. It has been shown that, on popular search spaces, optimizing hyperparameters can lead to a greater improvement in validation accuracy than optimizing the architecture [56]. However, it is much more difficult to run joint NAS and HPO than to perform these processes in isolation. In this joint setup, the search space becomes very complex, and different parameters have different effects depending on the available computational budget [56].

3.3 NAS goals

Neural Architecture Search aims to find a high-performing neural network architecture within a predefined search space under computational constraints such as limited training time, memory, or

computational budget. It aims to maximize a predefined performance metric for a fixed dataset and a training protocol.

3.4 Applications

NAS has been successfully applied in numerous domains beyond image classification. Although the majority of NAS work has been applied to image classification, there are numerous examples of successful applications in less-used domains. NAS has been extended to several neural network architectures, including graph neural networks (GNNs), generative adversarial networks (GANs), and Transformer models. It has also been applied to dense prediction tasks, such as semantic segmentation and depth estimation. Graph neural networks are neural networks that process data represented as graphs. Generative adversarial networks consist of a generator and a discriminator that are trained to compete with each other in a game-like setup. Dense prediction tasks require predictions for every pixel or spatial location in an image, such as semantic segmentation or depth estimation. NAS has also been applied to Transformer architectures used in language modeling and computer vision. Despite these extensions, image classification remains the dominant benchmark for evaluating NAS methods.

3.5 Task definition

The task in this study is formulated as a supervised learning problem with inputs x, targets y, a predictive function $f(x)$, and a loss function used to optimize the model parameters. The targets are real-valued amplitudes. The model is trained to predict $\log(y)$, which allows for comparing very small amplitude numbers with each other.

4 Background

4.1 The transverse field Ising model

The Ising model is a mathematical model of interacting spins that is widely used to study magnetism and phase transitions [11]. The transverse field Ising model (TFIM) extends the classical Ising model by introducing a magnetic field perpendicular to the interaction axis, allowing the spins to undergo quantum fluctuations.

In the one-dimensional Ising model, there are N spins which occupy equally spaced sites on a linear lattice. Each lattice site contains a spin variable, representing spin-down and spin-up states. In an ordered phase, neighboring spins tend to align. In a disordered phase, quantum fluctuations prevent long-range spin alignment.

The Hamiltonian equation defines the energy in the quantum system. Solving the Hamiltonian yields the ground-state wavefunction, whose amplitudes are the target values predicted in this work. This Hamiltonian has the form:

$$\hat{H} = \sum_{i=1}^{N-1} \sigma_i^z \sigma_{i+1}^z + \sigma_N^z \sigma_1^z - h \sum_{i=1}^{N} \sigma_i^x \tag{2}$$

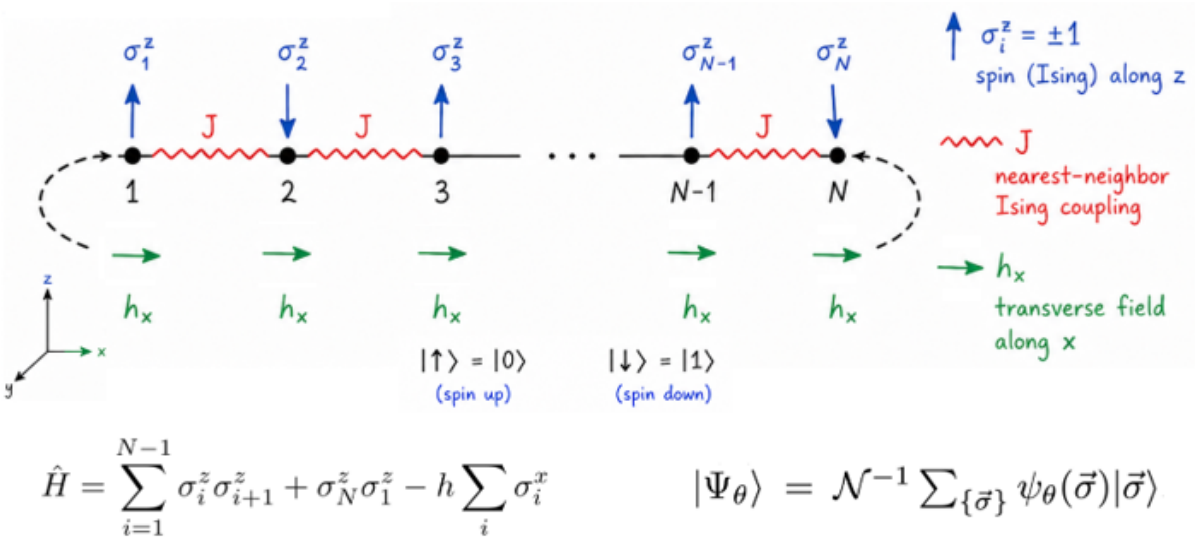


Figure 5: Explanation of the transverse-field Ising model (TFIM), superposition amplitude, and the Hamiltonian equation representing the ground state energy of the quantum system.

The parameter h in the equation represents the strength of the magnetic field that tries to flip the spins in a perpendicular direction. The $\sum_{i=1}^{N-1} \sigma_i^z \sigma_{i+1}^z$ term represents the local spin-spin interaction between neighboring particles in the system. The $\sigma_N^z \sigma_1^z$ element represents the interaction between the first and last spin in this system (periodic boundary condition). The chain of spins forms a closed loop in this setting, where the first spin and the last spin interact together. The $h \sum_{i=1}^{N} \sigma_i^x$ term represents the effect of the external magnetic field applied to and affecting all spins in the TFIM system. The last element of this equation (the external magnetic field) tries to change the direction of spins along the X-axis, while the first two terms (indicating local spin–spin interactions) try to align the spins along the Z-axis. The strength h of the external magnetic field controls a phase transition from ferromagnet to paramagnet with a critical point between those transitions at $h = 1.0$. The J value of coupling constant measures how strongly neighboring spins interact. In the setting introduced in [40], $J = 1$.

In this work, we consider the computational setting introduced by [40]. In this setting, there are N spins that hold a binary value—either 0 or 1. Each unique combination of spins creates a spin configuration that is mapped to a wavefunction amplitude, which is a real number.

4.2 The quantum many-body wavefunction

The wavefunction is a complex-valued vector in Hilbert space that encodes the state of a quantum system. The computational z-basis is the basis in which each spin is an eigenstate of the Pauli-z operator σ^z. The wavefunction can be represented as $|\Psi\rangle = \sum_{\mathbf{s}} C_{\mathbf{s}} |\mathbf{s}\rangle$ where $C_{\mathbf{s}}$ stands for amplitude of configuration $\mathbf{s}$. A system of N spins has a Hilbert space of dimension 2^N, with each basis vector corresponding to a distinct spin configuration.

4.3 Neural quantum states

This subsection discusses parameterization and model capacity of NQS. Neural quantum states are feedforward neural networks used to efficiently represent many-body quantum wavefunctions [13]. NQS aim to approximate the true wavefunction amplitudes $C_{\mathbf{s}}$ by a parametrized function

$$\psi_\theta(\mathbf{s})$$

where $\mathbf{s}$ denotes the full spin configuration, θ represents network parameters. The function

$$\psi_\theta(\mathbf{s})$$

returns the predicted wavefunction amplitude for configuration $mathbfs$. For example, storing a general wavefunction of 60 spins requires 2^{60} complex amplitudes, which is computationally infeasible. In many physical systems such as the transverse-field Ising model (TFIM), locality can be exploited to construct efficient representations. Locality in the TFIM model means each part of the system mainly interacts with its nearby neighbors. Based on the choice of the NQS architecture design, we can encounter substantial differences in performance [13]. NQS often lack direct interpretability, making it difficult to identify which correlations in the system are most strongly captured by the model.

4.3.1 Parameterizing quantum wavefunction

The ability of NQS to accurately represent a quantum wavefunction depends on the model capacity [40]. NQS provide a compact representation of high-dimensional quantum states despite the exponential growth of the Hilbert space, with an increasing number of spins. This deep learning paradigm operates in two regimes of parameters. One of them is the under-parameterized regime, when the model has fewer parameters than needed to perfectly approximate the data. In the regime when the model has just enough capacity, this point is called the interpolation threshold [47]. In the over-parameterized regime, the model has more parameters than needed to fit the data. In the setting studied in [40], models with a large number of parameters may overfit on the available training data, which corresponds to a bias-variance tradeoff. In the setting introduced in [40], the total number of trainable parameters in the model depends on the width W, depth of the architecture, input and output dimensions.

4.3.2 Double descent

Although neural quantum states were inspired by deep learning, it was not clear to what extent they share the same properties as other neural networks, such as the double descent phenomenon [40]. The double descent phenomenon has been observed on a variety of machine learning tasks [40]. In this phenomenon, while increasing the number of parameters in the network, the performance initially degrades until the interpolation threshold. After reaching this point, the performance improves again when the model becomes over-parameterized. The work [40] examines whether NQS exhibit the same double-descent behavior observed in conventional neural networks.

In [40], the double descent phenomenon was studied using three-layer feedforward neural networks to predict TFIM ground-state amplitudes (see Figure 6). In a setting introduced in [40], a spin configuration is a 1D chain of zeros and ones. In the under-parameterized regime, the number of

unique training configurations correlates with the network generalization property. The loss function used for this task is inspired by the Hellinger distance between two distributions over the entire Hilbert space [40]. The experiments in [40] showed that the interpolation threshold lies well beyond the Hilbert space dimension, which indicates NQS typically operate in the under-parameterized regime. As a consequence, increasing model size and following a "bigger is better" approach does not necessarily improve generalization, which motivates the exploration of alternative NQS architectural choices. Model performance was evaluated using the infidelity metric. The infidelity measures the disagreement between the predicted wavefunction and the exact ground-state wavefunction. Lower values indicate a more accurate approximation:

$$\mathcal{I} = 1 - \left|\langle \Psi_\theta | \Omega \rangle\right|^2 \tag{3}$$

4.3.3 Spin configurations

The goal of the NQS network in this study is to approximate the true amplitude for small system sizes (N = 12 spins) using a compact approximation of the wavefunction [40]. Each training example consists of a spin configuration as the input and its corresponding ground-state amplitude as the target value. For a system of $N = 12$ spins, the Hilbert space contains $2^{12} = 4096$ basis states (spin configurations). It was shown that the NQS generalization property depends on data composition (see Section 4.3.4). The amplitudes are non-negative. Their squared values correspond to the probability of observing each spin configuration. Since the wavefunction is normalized, the squared amplitudes over all amplitudes sum to one. In [40], rather than selecting configurations at random, the training set contains 75% of configurations with the largest amplitudes from the Hilbert space, as [40] found that this ordering improves generalization on the test set.

4.3.4 Dataset composition

The dataset composition affects the network's ability to learn wavefunction amplitudes. The [40] paper compares two dataset compositions: an amplitude-ordered split and a uniform split. Due to very small amplitudes that contribute little to learning in the uniform split, the test-loss minimum does not strongly depend on the number of unique training configurations [40]. For a small number of unique training configurations, it is easy for the network to memorize the dataset, but this leads to poor generalization for unseen configurations [40]. The amplitude-ordered training set contains the most probable configurations, providing more information about the probability distribution defined by the squared wavefunction amplitudes.

4.3.5 Loss functions

The choice of a loss function affects the learning performance. The mean squared error (MSE) loss function is one of the most popular loss functions in machine learning. Compared with the MSE, the Hellinger-inspired loss produced smoother and more stable optimization in the experiments [40]. Although Hellinger Loss provided smoother trainings, MSE loss produced better models with lower losses.

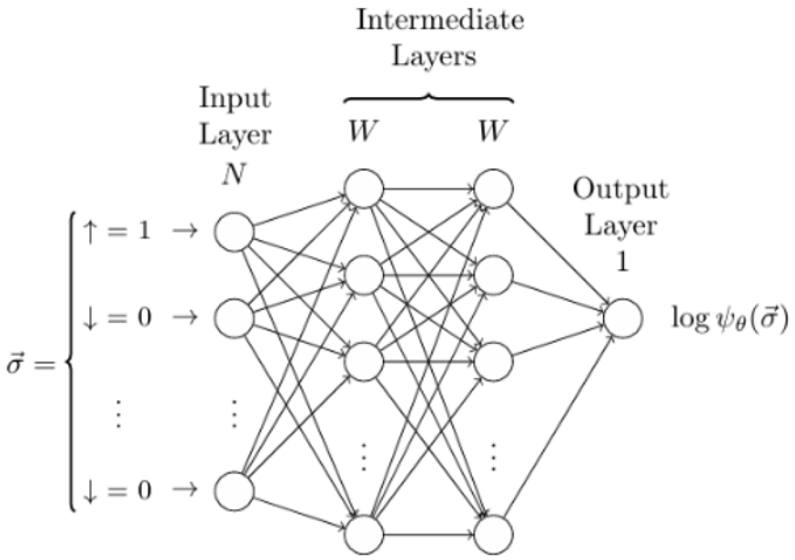


Figure 6: Neural quantum states (NQS) architecture where the input layer consists of a configuration of N spins. The intermediate (hidden) layers transform the input, and the output is an amplitude on a logarithmic scale. Figure taken from [40].

5 Related Work

5.1 Benchmarking

Neural architecture search (NAS) benchmarks provide standardized environments for evaluation of NAS algorithms. The benchmark components include a fixed search space, training pipeline, and evaluation protocol. Tabular NAS benchmarks provide precomputed evaluations for all possible architectures in the search space. NAS-Bench-101 was the first tabular NAS benchmark, released in 2019 [58]. Although [58] does not explain the origin of number 101, the most probable naming convention is analogical to the introductory course numbering at universities, e.g. 'Computer science 101', what is connected to the first foundational NAS benchmark [31].

5.2 NAS-Bench-101

5.2.1 Motivation

In the early stages of NAS research, architectures were optimized only for final test accuracy on CIFAR-10 or ImageNet. The search space and training pipelines differed across methods. This made it difficult to fairly compare NAS methods and reproduce reported results. Queryable NAS benchmarks have alleviated this issue by providing fixed and standardized evaluation components.

5.2.2 Search space

The search space of NAS-Bench-101 is cell-based, where each architecture is constructed from directed acyclic graphs (DAGs) representing computational cells. Each cell is stacked three times to form a full network. Each cell is constrained to at most 7 nodes and 9 edges, where each node

can take one of three operation types. NAS-Bench-101 creates a search space of 423k convolutional neural network architectures for image classification. The full search space was analyzed, requiring approximately 100 TPU-years of computation, yielding insights into architecture performance and NAS algorithms [58]. The architectures can be encoded in different ways, and an encoding choice can bias the search space and NAS algorithm performance [58]. Each architecture from this search space is trained multiple times (with different seeds) on the CIFAR-10 benchmark dataset.

5.2.3 Role in research

NAS-Bench-101 allows researchers to avoid repeatedly training architectures during search by using precomputed evaluation metrics, such as MSE or MAE on the test set. In the NAS-Bench-101 setting, architectures were originally trained under a fixed set of hyperparameters selected via coarse grid search. Each architecture was trained with three different seeds and four different training budgets (number of epochs). This architecture dataset is a mapping from architectures to their evaluation metrics. NAS-Bench-101 enables evaluation of NAS algorithms that aim to find high-performing architectures under specified constraints.

NAS-Bench-101 examines the influence of neural network operations and cell topology on the performance of CNNs and the relationship between number of parameters, training time, width, depth, and validation accuracy. The precomputed results in the NAS-Bench-101 table allow researchers to investigate the impact of various architectural choices on the performance of the network. The benchmark analysis showed that the best cell is not the most computationally intensive one. Although there is a positive correlation between these factors, this relation is not unambiguous. The search space exhibits locality, meaning that similar architectures tend to have similar performance.

5.2.4 Algorithms analyzed

The NAS-Bench-101 authors compared a diverse set of NAS algorithms to evaluate the usefulness of the benchmark. The algorithms analyzed include Reinforcement Learning, Bayesian Optimization Hyperband (BOHB), and SMAC. Each algorithm was compared against random search and evaluated based on 500 independent runs, achieving robust performance.

5.3 Other benchmarks and applications

NAS-Bench-101 served as a baseline and inspiration for a variety of subsequent benchmarks which extended the original work in several directions (see Sections 5.3.1, 5.3.2 and 5.3.3). The follow-up work includes extending the search space, modeling surrogate models, or increasing the number of application domains. While all benchmarks emphasize the importance of reproducibility of NAS research and the tremendous computational resources that traditional NAS requires, their search space, task range, and evaluation pipelines are substantially different. A surrogate NAS benchmark is a benchmark that uses a surrogate model to predict the performance of any architecture in the search space. NAS-Bench-1Shot1 simulates one-shot algorithms by defining a subset of the NAS-Bench-101 search space with a fixed number of nodes. NAS-Bench-201 is another tabular NAS benchmark precomputed on CIFAR-10, CIFAR-100, and ImageNet datasets. Surr-NAS-Bench-DARTS was the first surrogate NAS benchmark [56]. NAS-Bench-NLP consists of an LSTM-inspired

search space. NAS-Bench-ASR is a benchmark designed for automatic speech recognition. NAS-Bench-360 and NAS-Bench-Suite are benchmark suites that provide NAS benchmarks for a diverse set of problems, such as prosthetics control, protein folding, and astronomy imaging [56].

5.3.1 NAS-Bench-201

NAS-Bench-201 [14] also raises the reproducibility problem of NAS algorithms. Released one year after NAS-Bench-101, NAS-Bench-201 redesigned the original NAS-Bench-101 search space. The main difference between the two benchmarks is that NAS-Bench-201 has operations on edges, while NAS-Bench-101 has operations on vertices [14]. The search space of NAS-Bench-201 consists of all architectures generated by 4 nodes and 5 associated operation options, which results in 15625 neural candidates in total (substantially fewer than 423k in NAS-Bench-101). Contrary to NAS-Bench-101, NAS-Bench-201 supports a broader range of NAS algorithms, including parameter-sharing-based methods and network morphisms. Moreover, in NAS-Bench-201, there are no constraints on the number of edges [14]. Additionally, NAS-Bench-201 is evaluated on 3 different datasets and provides more diagnostic information than NAS-Bench-101 [14].

5.3.2 NAS-Bench-301

NAS-Bench-301 explores the idea of surrogate models for low-cost performance approximation [51]. NAS-Bench-301 targets the main limitation of NAS-Bench-101 and NAS-Bench-201, which is a consequence of tabular benchmarks, mainly extremely small architectural search spaces and high usage of computational resources [51]. Such tabular benchmarks exhaustively evaluate the entire search space where every possible architecture is trained and evaluated. This solution does not transfer to larger search spaces [51]. Instead of exhaustively evaluating the search space, NAS-Bench-301 utilizes surrogate models which enable evaluation of much larger search spaces such as DARTS containing up to 10^{21} candidate architectures. The NAS-Bench-301 authors showed that surrogate benchmarks can approximate architecture performance with high accuracy while utilizing only a small fraction of the computational cost of tabular benchmarks.

5.3.3 NAS-Bench-Suite

NAS-Bench-Suite collects multiple NAS benchmarks that evaluate NAS algorithms across diverse search spaces and tasks. The analysis demonstrated that conclusions reached on a single benchmark often fail to generalize across different search spaces and datasets. The paper highlights the importance of multi-benchmark evaluation [39].

5.4 Research gap

Up to this point, existing NAS benchmarks have focused primarily on classical machine learning tasks, such as image classification, natural language processing, and automatic speech recognition [39]. To the best of our knowledge, there is no established NAS benchmark for neural quantum states. This work addresses this gap by adapting the NAS benchmarking framework to neural quantum states. Instead of evaluating NAS methods on classical ML tasks, the proposed benchmark evaluates their ability to discover architectures that accurately represent quantum many-body wavefunctions.

6 Approach

Initially, the search space was defined by specifying all considered architectures. The objective is to identify which architectural choices, such as activation function, network width and depth improve NQS performance. Across the four datasets which represent different Hilbert-space regimes (see Section 6.2), the optimal architecture is identified based on different choices of activation functions, depths, and widths. The main methodology component is to create an automatic pipeline that generates architectures from a specified search space, trains each architecture, evaluates its performance, and stores the results in a structured format. The experiments are analyzed to characterize the search space and identify architectural features that correspond to high-performing NQS.

6.1 Research questions

The main research question that will be explored in this project is: *Which MLP architectural features improve the performance of neural quantum states?* This question is subdivided into 2 sub-questions:

How does the interaction strength between particles influence the choice of optimal MLP architecture?

The goal of this study is to examine the relationships between particles in different regimes that are influenced by spin-spin interactions and the transverse magnetic field. The specific interactions between those components influence which architectures are the most suitable for discovering those relationships. For example, spins can form clusters or can be concentrated around a single point based on their amplitude.

Which architectural choices are valuable for discovering relationships in quantum systems?

This work constructs a tabular benchmark to identify which architectural choices are associated with strong neural quantum state performance. The choices include network depth, widths, and the choice of an activation function. The goal of this study is to find a subset of those choices that is the most beneficial for approximating the wavefunction amplitude in quantum systems. The aim of this study is to investigate the architectures with the highest evaluation metrics on the test set and report their results.

6.2 Datasets

This study uses four datasets, each corresponding to a different value of the external magnetic field h. The h values are $h = 10^{-6}$, $h = 0.5$, $h = 1.0$, and $h = 2.0$. These values span the most relevant physical regimes of the TFIM, ranging from the nearly classical regime (with negligible quantum effects) to the regime with strong quantum effects. The selected values cover the ferromagnetic phase, the quantum critical region, and the paramagnetic phase. Each dataset consists of one-dimensional spin configurations of 12 spins. Each spin holds either 1 (up) or 0 (down) value. Each dataset is saved in a .csv file, which contains 2^{12} rows (the entire Hilbert space). Each row contains a spin configuration together with its normalized ground-state wavefunction amplitude. The squared magnitude of the amplitude gives the probability of observing the corresponding spin configuration. The normalized squared magnitudes of the amplitudes over all configurations sum to one. The datasets used for this study were provided by the supervisor. They were generated using density

matrix renormalization group (DMRG), which is a numerical algorithm for the efficient truncation of the Hilbert space. The DMRG algorithm is applied in the calculation of static, dynamic, and thermodynamic quantities in quantum systems [48].

	config	amplitude	probability
1	000000000000	0.8497603620873733	0.7220926729748636
2	100000000000	0.10799532347145337	0.011662989891703845
3	010000000000	0.10799697807054348	0.011663347272369447
4	110000000000	0.027465070815232875	0.0007543301148857567
5	001000000000	0.10799908754280058	0.0116638029100775

Table 1: Visualization and demonstration of the first 5 entries from a 1D TFIM dataset where $h = 0.5$. Each unique spin configuration is mapped to its corresponding amplitude and probability.

6.2.1 Explanation and reasoning why particular h values were chosen

As h increases, the transverse magnetic field becomes increasingly dominant over the spin-spin interaction. The selected h values span different quantum regimes and the datasets span different regimes of the phase diagram. For the dataset where $h = 10^{-6}$, it represents the classical Ising limit where there are nearly no noticeable quantum effects present. In the classical Ising model, the nearest-neighbor interaction dominates over the transverse magnetic field, present in the Hamiltonian equation (see Equation 2). For the model where $h = 0.5$, the transverse magnetic field becomes dominant, but we are still in the antiferromagnetic phase. For $h = 1.0$, the quantum system is at the critical point that separates the antiferromagnetic and paramagnetic phases. The transverse field induces quantum fluctuations in the x-direction (see Figure 5). At the highest value of $h = 2.0$, the transverse magnetic field dominates over the spin-spin neighboring interactions.

6.2.2 Amplitude distribution

The Figure 7 shows how similar the amplitude distributions of datasets in particular regimes are to the uniform distribution. As in Figures 10, the ranges of amplitude values on a logarithmic scale differ for particular configurations. For example, the logarithmic amplitude range in the regime where $h = 10^{-6}$ is from -140 to 0, indicating that there are only a few amplitude configurations of spins with a significant probability of occurring. In the $h = 2.0$ regime, the logarithmic amplitude range varies from -5 to approximately -3.5, demonstrating that this distribution is much closer to the uniform distribution, where the amplitude values are more comparable to each other. However, the relationship between the h value and the logarithmic amplitude range is neither obvious nor linear, as can be observed in Figure 7, where the $h = 2.0$ amplitude distribution achieves nearly perfect proximity to the uniform distribution. Figures 10 suggest that increasing proximity to the uniform distribution increases the complexity of the distribution, which is a major bottleneck for NQS models.

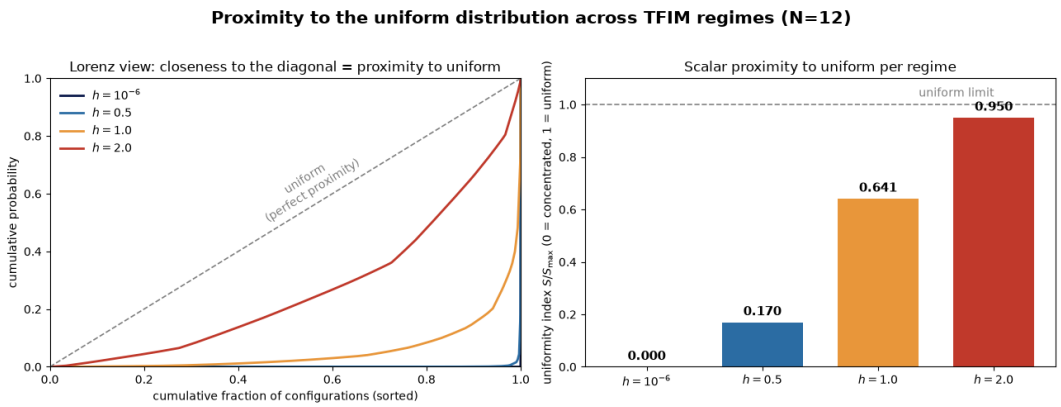


Figure 7: The proximity of amplitude distributions to the uniform distribution for datasets in different regimes.

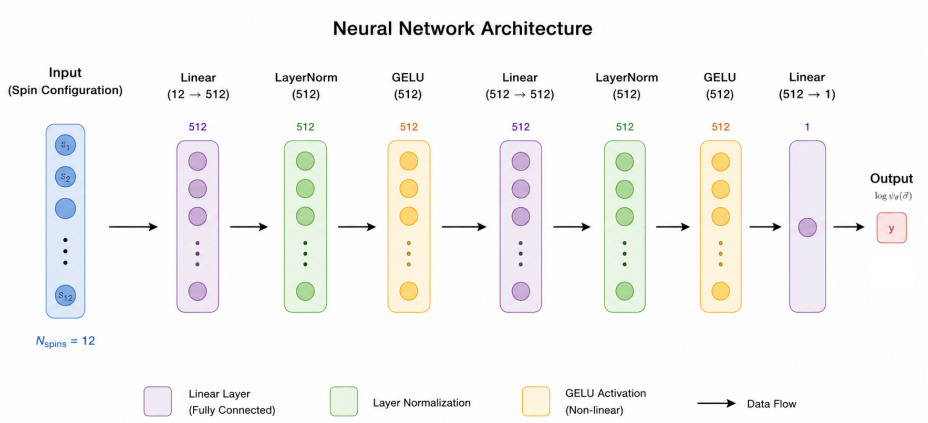


Figure 8: A visualization of one of the architectures from the search space. In this setting, GELU activation function is used, the width in each hidden layer is 512 and there are 2 hidden layers.

6.3 Search space

6.3.1 Overall network structure

Similarly to [40], compact feedforward architectures with up to three hidden layers are trained and evaluated. The search space is defined by the ranges of network depth, width and the set of allowed operations. Unlike [58], our architecture topologies consist of sequential layers, with 3-element modules. Each such module is constrained to a single linear layer, layer normalization, and an activation function (see Figure 8).

6.3.2 Activation functions

Activation functions introduce non-linearity, and the selection of activation function used can influence neural network optimization performance [50]. In the current setup, 3 different activation functions are used. For wide networks (network width >= 432), activation functions help to mitigate training instabilities during the training process [40]. The selected activations for this

study represent different nonlinear properties and commonly used activation functions for training neural networks.

The *ReLU* activation function is defined as $ReLU(x) = max(0, x)$. ReLU is one of the most commonly used activation functions, however, ReLU can suffer from the dying activation problem, where neurons output zero for all inputs and stop updating during training.

The GeLU activation function is the default activation function used in this study. Its smooth shape serves as a smoother differentiable alternative to ReLU. This activation function is represented by a Gaussian distribution function $\text{GeLU}(x) = x \cdot \Phi(x)$

The hyperbolic tangent activation function is zero-centered and transforms the input into a range $[-1, 1]$. Mathematically, this function is defined as $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{\sinh(x)}{\cosh(x)}$.

6.3.3 Network widths and depths

Network width refers to the number of neurons in a hidden layer. The network widths range from 16 to 8192. For some architectures, the width is fixed for every layer. Some architectures use the same width for every hidden layer, whereas others allow the width to vary between layers. Given a set of possible widths and a set of network depths, all possible architectures are generated. The search space includes network depths ranging from 1 to 3 hidden layers, similarly to the setting introduced in [40]. Each hidden layer block consists of a fully connected linear layer, layer normalization, and an activation function (see Figure 8 for illustration).

6.4 Evaluation metrics

To effectively evaluate the trained machine learning models, a wide range of regression statistical metrics are used, each with its own unique strengths and weaknesses. R^2 coefficient of determination is evaluated on the train, test, and validation sets and saved into evaluation metrics:

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \tag{4}$$

.

Mean squared error metric was used as a loss function for most training runs. MSE is defined as the average of the squared differences between the actual and predicted outputs. This calculation is performed for all instances in the dataset, and the resulting values are averaged:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2 \tag{5}$$

.

Mean absolute error (MAE) is one of the evaluation metrics used as a systematic result saving. MSE describes, on average, how far off your model's predictions are from the actual values. Due to the absolute operations, too high and too low errors are treated equally (the error sign does not matter):

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{6}$$

.

The Hellinger distance measures similarity between two probability distributions and can be used as a loss function in quantum systems:

$$\mathcal{L}_{\mathcal{H}}(\psi_\theta, \Omega) = \frac{1}{\sqrt{2}} \sqrt{\sum_{\vec{\sigma} \in \mathcal{D}_{\text{Train}}} \left(\psi_\theta(\vec{\sigma}) - \Omega(\vec{\sigma})\right)^2} \tag{7}$$

.

The infidelity metric, which is defined as one minus fidelity, measures the dissimilarity between the predicted and target quantum states:

$$\mathcal{I} = 1 - F \tag{8}$$

.

For each architecture, the number of trainable and non-trainable parameters is recorded. For each training run, the number of non-trainable parameters is zero, which indicates that every parameter (weight or bias) is updated during training and there are no frozen weights or other fixed parameters included in the parameter count.

The total training time (in seconds) for each architecture was measured and stored to analyze the relationship between training time and performance. Initially, all models for each regime were trained in the same loop where each regime had its own dataloader and a model. However, this approach led to saving the same training time for each model, what resulted in incorrect training time for models trained in one training loop. Training each model one after another solved the issue.

7 Experiments

The conducted experiments evaluated architectures across the predefined search space to identify combinations of architectural parameters that achieve strong predictive performance for the regression task. Each architecture was trained and evaluated separately on each dataset what corresponded to a different value of h.

7.1 Hyperparameter optimization

In the current setup, a fixed set of hyperparameters is chosen for each architecture and the hyperparameter configuration was inspired by [40]. However, NAS-Bench-101 creates a NAS-Bench-HPO, where the architecture and hyperparameters are jointly optimized (including learning rate and batch size) [58], [26]. To ensure reproducibility and a fair comparison, all architectures were trained using the same hyperparameter configuration.

7.2 Training

The training setup in this work follows a modular design where architecture generation, model training, evaluation, and result storage are implemented as separate components. To enable training of a large number of architectures, a configuration file determines ranges of different parameters to set during training. During training of all regimes, there is one training loop where dataloaders

and models of individual regimes are updated. For every training epoch, gradients are computed, the backpropagation algorithm is run, and parameters are updated.

The massive architecture training in this setup is utilized using loops with parameter ranges (see a simplified pseudo-code Algorithm 1). A configuration module defines constant parameters used in the training process, such as batch size, network width, and depth. The parameters distinguish architectures with fixed and changing widths in subsequent layers.

The configuration module enables a single architecture to be trained with a specified set of parameters. To enable training of many architectures in one script execution, there is a controller script that iterates over all parameter configurations from the search space and launches the corresponding training runs. In the experiments run on many architectures, each architecture is executed as an independent training run where the configuration for each architecture is passed with environment variables. Each architecture and its corresponding experiments are isolated from the rest of the architectures.

Algorithm 1 Simplified pseudo-code for automated training multiple architectures in a single run

```
for each training regime r ∈ regimes_combinations do
    for each epoch count e ∈ epoch_range do
        for each hidden layer count h ∈ hidden_layers_range do
            for each activation function a ∈ act_functions_range do
                for each network width w ∈ widths_range do
                    Configure architecture (h, a, w)
                    Generate and train the corresponding model
```

The actual training of architectures is performed by the main training module. Each regime gets its own dataloader, optimizer, model, and scheduler (if exponential learning rate is turned on). In each epoch, training and validation losses are saved, which are later used by the result-processing module to create a structured directory hierarchy and compute the evaluation metrics.

7.2.1 ALICE

This work was performed using the compute resources from the Academic Leiden Interdisciplinary Cluster Environment (ALICE) provided by Leiden University. This is a high-performance computing facility which is available for Leiden Institute of Advanced Computer Science (LIACS) and Leiden University Medical Center (LUMC) researchers [49]. The computational requirements of evaluating the complete tabular benchmark made training on a standard personal computer impractical. The experiments were executed on GPU-equipped compute nodes provided by the ALICE high-performance computing facility. This study utilizes GPU-equipped nodes with computational resources allocated through SLURM job scripts.

7.2.2 Scheduler

In the current setup, the scheduler is chosen based on the highest layer width in each neural network. Following [40], a width threshold of 432 neurons was used to distinguish between small and large networks. The Adam optimizer is used in each architecture. For small networks ($W < 432$), $ExponentialLR$ scheduler is used. For larger networks ($W \geq 432$), a warmup ($LinearLR$) phase is

added to *ExponentialLR*. Instead of updating after every iteration, the scheduler is updated every 50 epochs to improve stability.

In the scheduler, the hidden layer width is used to decide the schedule type - for small and large networks. For small networks ($W < 432$), the optimizer is Adam with exponential decay. In case of large networks ($W >= 432$), a linear warmup is added. This scheduler sets parameters such as decay rate, initial learning rate, and learning rate peak. Moreover, a number of transition and warmup steps is defined.

The key idea behind this scheduler is that large networks need more complex training, which includes a warmup phase. The learning peak indicates the maximum learning rate used in training. The initial learning rate is the starting learning rate used in the warmup phase in large networks. The transition steps parameter means the number of optimizer steps necessary to update the scheduler. The number of warmup steps indicates the total duration of the warm-up phase for large models. The decay rate is the fraction of the previous learning rate that is saved in the next step (with decay rate 0.99, the learning rate decreases by 1% every optimizer step).

7.2.3 Results Storage and Organization

The results are categorized into subfolders for automated training of different architecture configurations (see Figure 9). Each training run is stored in a timestamped folder. The subfolder names encode the values of the corresponding architecture parameters. This setup facilitates reproducibility, automated experimentation across multiple parameter configurations, and comparison between different architectures.

The results-saving script computes and records evaluation metrics from the model predictions and corresponding ground truth values and saves them in a `.csv` file, along with a global CSV file containing the results of all architectures trained so far. A dictionary file defines the columns recorded for each architecture, including the number of epochs, batch size, and regression metrics such as MSE. In addition to the global `.csv` file, each training run has its own timestamped folder, which contains all generated plots, including comparisons of predicted and ground truth amplitudes, as well as the training and validation loss for each epoch. To facilitate reproducibility and further analysis, the predicted and ground truth amplitudes are saved as separate `.txt` files.

8 Results

The preliminary experiments show that increasing the external magnetic field strength is associated with reduced prediction performance (see Figures 10a, 10b, 10c, and 10d). The GeLU activation function consistently achieved the best performance, which may be attributed to its smoothness matching the smooth nature of the quantum wavefunction. Increasing network width does not necessarily lead to improved performance. Feedforward neural networks with width 16 and depth 3 can approximate the quantum wavefunction well, even for the critical regime $h = 1.0$, where the entanglement structure is more complex (see Figure 10c).

8.1 Result comparison over different regimes

Overall, predicting a quantum many-body wave-function using parameterized NQS remains a challenging task across most regimes in this study. The plots in Figure 10 show the performance of

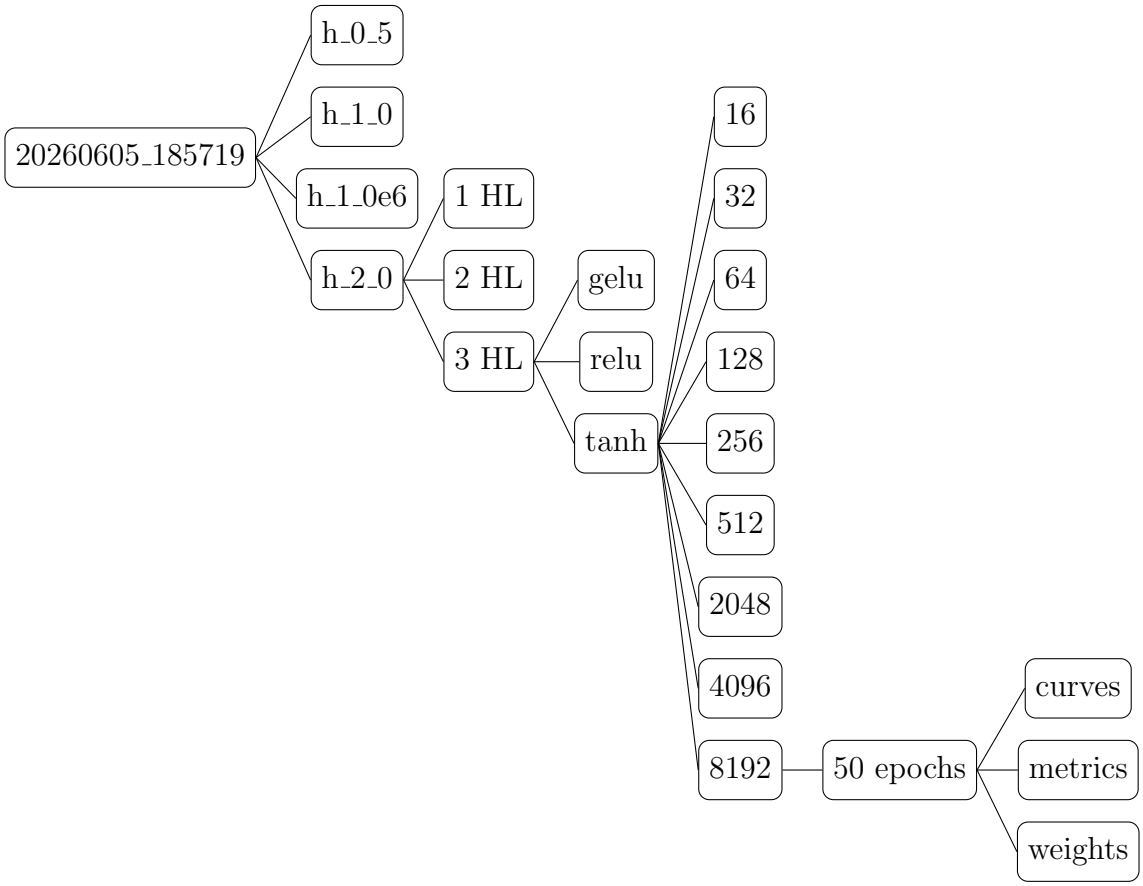
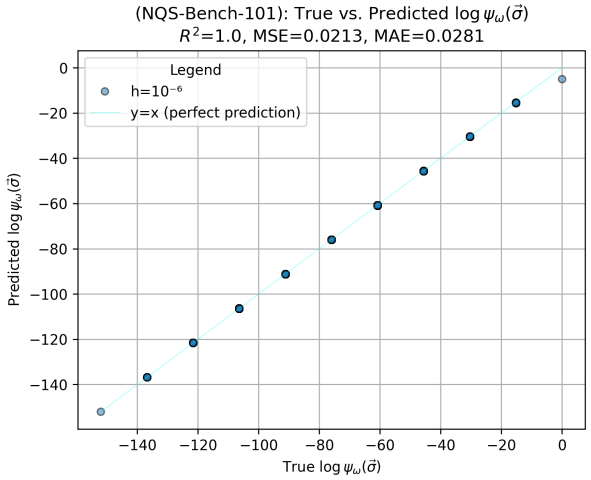
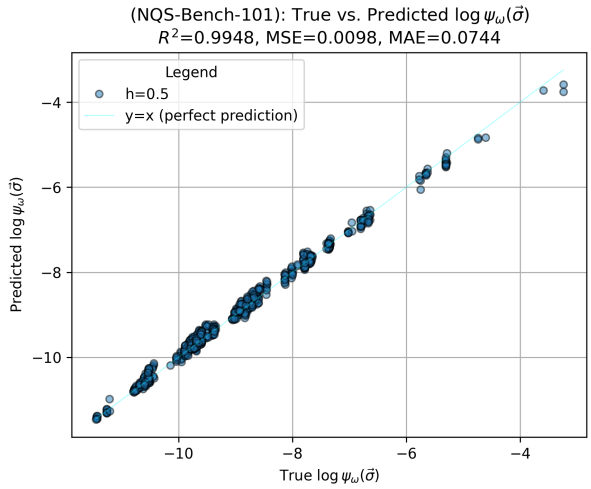


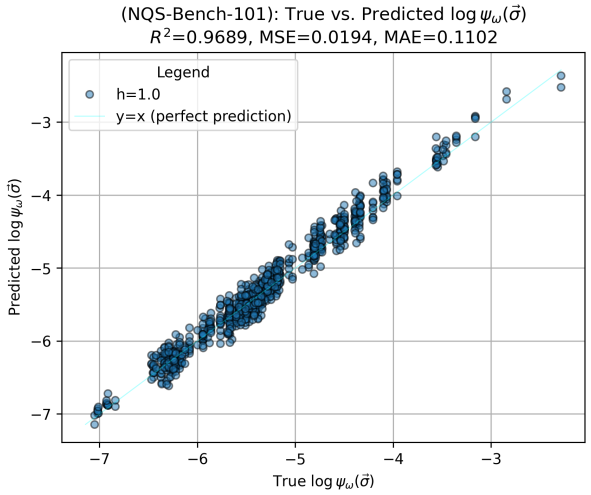
Figure 9: A schematic representation of the directory tree for a single training run. The tree is shown for illustrative purposes only; the actual parameter ranges may differ.



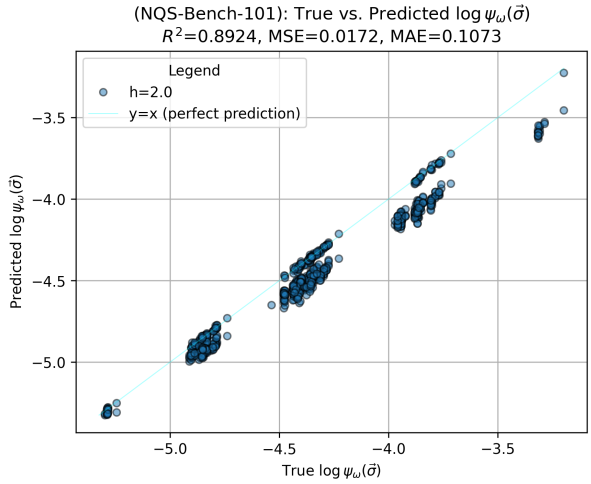
(a) Regression results for regime $h = 10^{-6}$. In this regime, quantum effects are negligible, and the system is nearly identical to the classical Ising model. Model configuration: 2 hidden layers, width 16 in each hidden layer, 500 epochs, GeLU.



(b) Regression results for the regime $h = 0.5$. The system is in the ferromagnetic phase, where quantum effects are significant. The external magnetic field tends to flip the spins, making the prediction task slightly more challenging than for $h = 10^{-6}$. Model configuration: 2 hidden layers with width sequence 8 and 256, 500 epochs, tanh.



(c) The regime $h = 1.0$ corresponds to the quantum critical point of the TFIM, where entanglement between spins is strongest. Model configuration: 3 hidden layers with sequence 16, 32 and 256, 60 epochs, GeLU.



(d) The regime $h = 2.0$ is in the paramagnetic phase. The strong external magnetic field tends to align the spins with the field, making the prediction task more challenging than in the weak-field regime. Model configuration: 1 hidden layer 32, 500 epochs, GeLU.

Figure 10: Predicted versus true values for different transverse field strengths h.

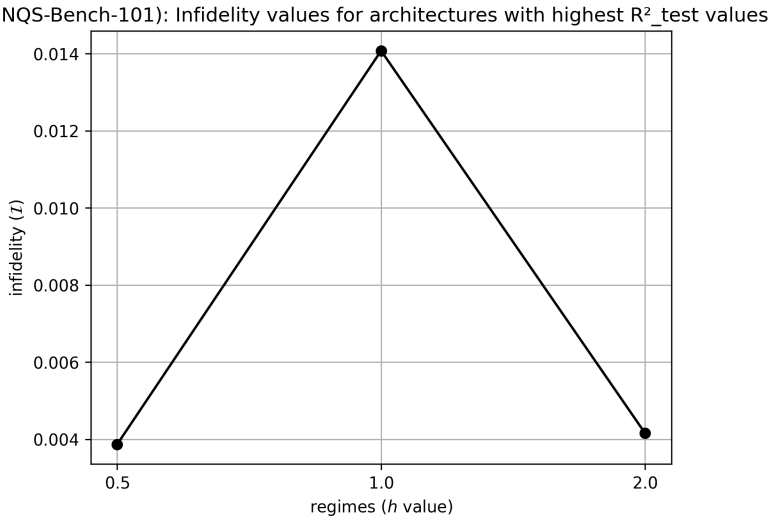


Figure 11: The infidelity metric for different regimes. High infidelity means that the learned wavefunction is far from the true wavefunction. Among the evaluated regimes, $h = 1.0$ achieves the lowest infidelity.

the best models for each regime present in the full architecture dataset. Although some models perform considerably well in each regime (see Figures 11 and 10), the Figure 10a in $h = 10^{-6}$ regime achieves the highest performance, with nearly perfect R^2, MSE and MAE evaluation metrics. Although this regime might indicate near-perfect prediction, the system exhibits almost no quantum effects and is close to the classical Ising limit. The variance in prediction performance can be understood by examining the distribution of wave-function amplitudes. The Figures 11 and 10 indicate that for lower `h` values, `h` concentrates the probability mass around a smaller number of configurations, creating a distribution that is easier for the NQS models to learn. In Figure 10b, the transverse magnetic field begins to cluster configurations, and the scale of x and y axis varies significantly in comparison to Figure 10a. With increasing `h`, the logarithmic amplitude range becomes lower and lower. In Figure 10a, the amplitude in a logarithmic scale ranges from -140 to 0 (suggesting amplitudes which vary in orders of magnitude), while in Figure 12b, the range of amplitude in a logarithmic scale ranges from -10 to -4. In Figure 10d shows the lowest range of amplitudes on a logarithmic scale (-5 to -3.5), suggesting the presence of higher amplitude values, and the amplitude distribution is more similar to the uniform distribution. The ratio between the nearest-neighbor interactions and the external magnetic field determines the behavior of the spin configurations and influences their representation in the embedding space. As `h` increases, the clusters corresponding to different spin configurations become more separated in the embedding space. The ratio between `h` value and local spin-spin interaction influence how spins are represented in the embedding space. The Figure 11 demonstrates that for infidelity metric measured across the entire space, the models' performance in $h = 0.5$ and $h = 2.0$ regimes is comparable, while for $h = 1.0$, the infidelity is higher, meaning that the probability of identifying the predicted quantum state as the true quantum state is lower. The results in Figure 11 demonstrate that for the best models in the architecture dataset, the relation and complexities in the quantum critical phase at $h = 1.0$ were the hardest to learn. The infidelity metric shown in Figure 11 is dependent on the model performance, rather than directly on the amplitude probability structure in particular

regime datasets.

8.2 Architecture dataset characteristics

The architecture dataset was constructed to systematically investigate how different architectural choices influence NQS performance. The resulting dataset contains 14,905 trained architectures, with each record corresponding to one evaluated architecture. Although smaller than the NAS-Bench-101 dataset [58], which was generated using a larger search space, more training budgets, and multiple training seeds, the proposed dataset is designed to provide controlled comparisons of architectural choices rather than an exhaustive evaluation of the entire search space. The dataset was collected incrementally during successive project stages, where increasingly larger portions of the search space were evaluated.

The search space consists of feed-forward neural networks with one, two, or three hidden layers for each evaluated NQS regime. Hidden layer widths are selected from the set $16, 32, 64, 128, 256, 512, 2048, 4096$. To gradually increase the complexity of the search space, three search space cases were defined based on possible choices of activation functions and hidden layer width sequences.

Case 1 forms the baseline search space, where all hidden layers share the same width and use the GeLU activation function. Case 2 increases architectural flexibility by allowing different widths in each hidden layer while keeping the activation function fixed. Finally, Case 3 introduces the highest level of flexibility by allowing each architecture to use one of three activation functions (ReLU, GeLU, or tanh).

Besides architectural complexity, training duration represents another factor that may influence the observed model performance. To investigate the effect of substantially longer training, a subset of 162 Case 1 architectures was additionally trained for 500 and 1000 epochs. This experiment separates the influence of training duration from architectural design.

Most architectures were trained for 30 and 60 epochs to balance computational cost with sufficient training quality. The majority of the dataset (10,512 architectures) was trained using 30- and 60-epoch budgets, while the remaining records correspond to additional training configurations collected during earlier project stages, including extended Case 1 experiments.

Overall, the dataset provides controlled variation in both architectural configuration and training budget, enabling systematic analysis of their influence on NQS performance.

8.2.1 Full architecture dataset

Table 2: Summary of selected statistics of the three TFIM regimes. There are 207 architectures not included in the table which belong to the $h = 10^{-6}$ regime.

	h=0.5 (n=7024)			h=1.0 (n=3807)			h=2.0 (n=3857)		
Statistic	R^2	MAE	Inf.	R^2	MAE	Inf.	R^2	MAE	Inf.
Mean	-2.379	1.428	0.244	-3.120	1.067	0.217	-6.504	0.669	0.086
Median	0.699	0.613	0.128	0.213	0.583	0.137	0.076	0.322	0.050
Variance	64.08	3.86	0.047	61.53	1.18	0.024	292.63	0.683	0.004
Skewness	-2.75	2.25	0.90	-2.45	1.95	0.40	-2.85	2.23	0.87

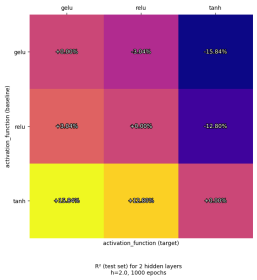
Overall, the full architecture dataset exhibits highly heterogeneous performance distributions. Standard machine learning metrics show strong sensitivity to extreme outliers, reflected by high variance and skewness values. In contrast, quantum evaluation metrics exhibit more concentrated distributions and lower sensitivity to outliers than R^2, MSE, and MAE, suggesting that they provide a more stable evaluation of quantum-model performance. Although the performance on evaluation metrics varied between regimes, all regimes contained models with relatively good performance. Since the number of evaluated architectures differs between regimes, aggregated statistics should be interpreted carefully, as they may be influenced by the relative contribution of each regime. As epoch budget and training time increased, the performance variance between experiments was lower, indicating the presence of more architectures which converge to a similar performance (see Figure 15). One possible reason for the occurrence of extreme outliers are architectures trained for a few epochs for testing and debugging purposes in early project stages or during implementation of a new feature.

The above-mentioned performance trend is visible in the R^2. The R^2 metric is the only metric in this comparison with a left-tail distribution instead of a right-tail distribution, since higher R^2 values are more desired. The extreme R^2 outliers correspond to negative values, indicating architectures performing worse than the baseline prediction.

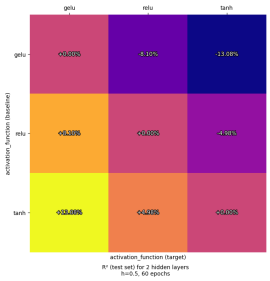
Although the $h = 0.5$ architecture dataset R^2 metric is also left-skewed (see a summary of selected metrics in Table 2), the mean and median are -2.37 and 0.70, respectively. The $h = 1.0$ architecture dataset contains more extreme and varied R^2 values than the $h = 0.5$ architecture dataset. The $h = 2.0$ architecture dataset presents the highest variance level across all regime datasets. The R^2 statistic in the extended Case 1 dataset distribution is fairly normal.

Although the MSE metric is expressed in a different scale than R^2, the MSE metric shows similar trends in the data of trained architectures as R^2. However, this difference is more visible (MSE full architecture dataset mean 30.44 and median 0.48). The median performance differences between training and test sets are higher than for R^2. The skewness=13.57 and kurtosis=193.57 of training and validation performance show that the MSE metric is more sensitive to extreme outliers than R^2. The interesting observation for MSE is that the MSE test set performance is higher than the validation set. The MSE performance for the $h = 0.5$ architecture dataset mean and median equals 6.40 and 0.57, respectively, with variance and standard deviation equal to 229.66 and 15.15, respectively.

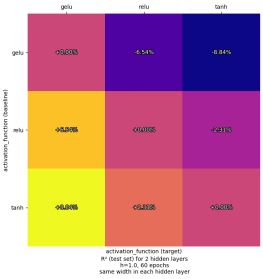
8.2.2 Activation functions vs. performance



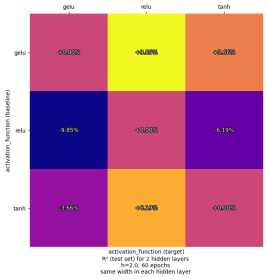
(a) Comparison on the extended Case 1 architecture dataset with $h = 2.0$, using architectures trained for 1000 epochs (81 architectures).



(b) Comparison on the full architecture dataset with $h = 0.5$, using architectures trained for 60 epochs.



(c) Comparison on the full architecture dataset with $h = 1.0$, using architectures trained for 60 epochs.



(d) Comparison on the full architecture dataset with $h = 2.0$, using architectures trained for 60 epochs.

Figure 12: Aggregated impact of changing the activation function, evaluated with the R^2 metric on the test set for different regimes. Only architectures with 2 hidden layers are considered and each architecture has a fixed width in each hidden layer.

Across most regimes, changing the activation function has a measurable but relatively modest effect on performance compared with changing hidden-layer widths. This trend is observed in Figures 12b, 12c, and 12a, where GeLU generally provides the highest average performance.

Figure 12b shows the average R^2 effect of replacing one activation function with another. Replacing one activation function in all hidden layers of a particular neural network leads to higher variability than replacing one hidden layer width with another (see Figure 17), since no single activation function consistently achieves the highest performance across all regimes. There is no uniform-similar average R^2 performance change for a particular activation function, which is also emphasized by different colors in each activation function heatmap column. Figure 12b presents a similar trend in performance as Figure 12a and Figure 12c, where replacing the GeLU activation function with tanh leads to the highest average R^2 performance drop. Figure 12b presents a gradual performance trend drop while replacing each activation function with GeLU, ReLU, and tanh, respectively, which shows the ranking of activation functions that lead to the highest average performance. This activation function performance order is also visible in Figure 12a and Figure 12c, but it is less symmetric than in Figure 12b. The performance in all four heatmaps suggests that the average performance difference while changing the activation functions is less sensitive than changing network widths (see Figure 17) to changing the value of the transverse magnetic field

h, since only the highest regime leads to a significant change in the performance trend between different activation functions. However, there is no clear relationship between how the strength of the external magnetic field affects the magnitude of change while changing the activation functions, since the highest full $h = 2.0$ dataset does not lead to the highest average performance change.

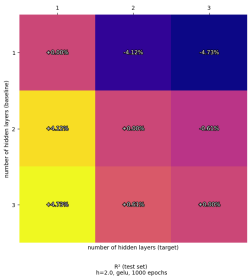
A similar activation-function ranking is observed in both the $h = 0.5$ and $h = 1.0$ regimes, where GeLU achieves the highest average performance, followed by ReLU and tanh (Figures 12b and 12c). Even though the trend in performance is similar, the average difference between those activation functions is smaller, as the change in average performance between ReLU and tanh is less than 2% (see Figure 12). However, the differences between the three above-mentioned activation functions are different. The average performance of ReLU and tanh varies by only a few percent, while GeLU is outstanding, giving a nearly 9% increase.

For the highest value of the transverse magnetic field, there is a totally different trend in the data, as the performance ranking of activation functions is different than in the rest of the activation function heatmaps. For a higher h value, the trend is contrary, with the worst-performing GeLU activation function. As the highest difference in the rest of the regimes was between GeLU and tanh, the difference in $h = 2.0$ is switched into ReLU and GeLU, indicating that the dominance of the transverse magnetic field has an influence on which activation function to choose and the choice is not obvious.

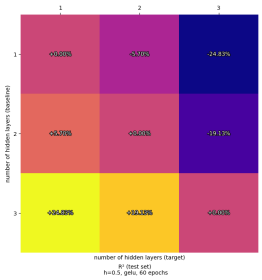
Figure 12a shows the performance of the "extended Case 1" dataset trained in $h = 2.0$. Even though it was observed that Figure 12d shows a totally different trend in the data than the rest of the regimes, the fraction of the search space present in "extended Case 1" leads to a similar trend as $h = 0.5$ and $h = 1.0$. For the heatmap in Figure 12a, the magnitude of change is the highest, with nearly 16% average R^2 performance drop while replacing GeLU with tanh. The average performance between GeLU and ReLU is similar. In the $h = 0.5$ and $h = 1.0$ regimes, the performance difference was observed between ReLU and tanh.

Overall, activation-function choice influences NQS performance, but its effect is generally smaller than that of hidden-layer width. GeLU provides the strongest average performance in the $h = 0.5$ and $h = 1.0$ regimes, whereas the $h = 2.0$ regime exhibits a different ranking in which ReLU performs better. The extended Case 1 experiment suggests that this reversal is not universal and may depend on the specific subset of architectures considered. Consequently, no single activation function is optimal across all regimes, indicating that activation-function selection should be considered jointly with the physical properties of the target system. Therefore, activation-function performance depends on both the physical regime and the architectural subset under consideration, rather than on either factor alone.

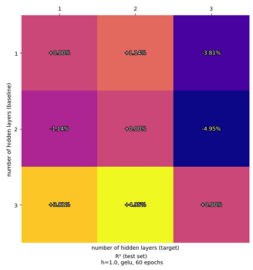
8.2.3 Network depth vs. performance



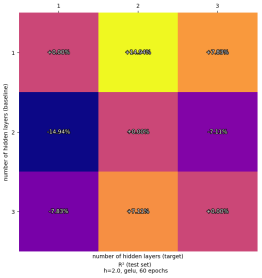
(a) Comparison on the extended Case 1 architecture dataset with $h = 2.0$, using architectures trained for 1000 epochs (81 architectures).



(b) Comparison on the full architecture dataset with $h = 0.5$, using architectures trained for 60 epochs.



(c) Comparison on the full architecture dataset with $h = 1.0$, using architectures trained for 60 epochs.



(d) Comparison on the full architecture dataset with $h = 2.0$, using architectures trained for 60 epochs.

Figure 13: Aggregated impact of changing the number of hidden layers using the GeLU activation function, evaluated with the R^2 metric on the test set for different regimes. The width of each hidden layer is allowed to vary independently.

Figure 17 shows the aggregated impact on the R^2 test set metric of replacing one hidden layer width with another, where only architectures with 2 hidden layers and the same width in each hidden layer, trained over 60 epochs, are considered due to the exponential growth of the number of hidden layer width combinations. Across all regimes, increasing network depth does not consistently improve NQS performance. Instead, the preferred depth depends on the physical regime, although shallow architectures generally remain competitive.

Figure 13b demonstrates the heatmap of changes for replacing network depth. For each depth heatmap, all networks with a certain number of hidden layers are considered (either the same number of neurons in each hidden layer or different width sequences). It can be observed that different network depths perform differently and that increasing the complexity of the architecture by increasing the depth does not, on average, lead to performance improvement. The interesting observation for the $h = 0.5$ regime is that the trend is totally opposite - shallow neural networks with only one hidden layer achieved, on average, the highest results, with a disproportionately lower difference for networks with three hidden layers than for architectures with either one or two

hidden layers. All heatmaps show an inconsistent trend in the magnitude of differences between different regimes, since the heatmap with the lowest h value achieves the highest magnitude of changes when replacing network depth one with three.

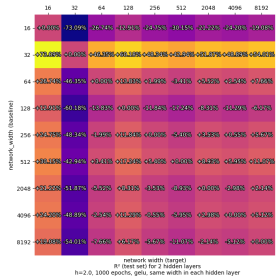
In the $h = 1.0$ regime in Figure 13c, the models in the TFIM critical phase show uniform-similar model performance for different depths and a different order of the average-best network depth. In the TFIM critical phase, the highest-performing network depth is two, while networks with three hidden layers still perform the worst on average. The results in Figure 13c indicate that, for the critical phase, choosing the right network depth is less significant, showing the importance of other network architectural choices to discover high-performing architectures. Changing the value of the external magnetic field changes the relation between performance for different network depths. In $h = 1.0$, the highest-performing architectures have two hidden layers. Architectures with one hidden layer remain competitive, but the performance differences are considerably smaller than in the $h = 0.5$ regime.

Increasing the h value to 2.0 shows how the importance of network depth changes, as shallow neural networks become, on average, the worst-performing architectures. The depth in this study corresponds to the number of hidden layers, and there are either one, two, or three hidden layers in each neural network architecture. Figure 13d shows that the relation between network depth and the transverse magnetic field is not obvious, as increasing the h value does not lead to reversing the relation (the trend of network depth and performance is not opposite, the variability of changes is not the highest, and networks with three hidden layers are not the best-performing architectures). Because only values up to $h = 2.0$ were considered, it remains unclear whether this trend continues or reverses at larger field strengths. Additional experiments would be required to answer this question. The observed differences may indicate that the relationship between network depth and performance depends on both the magnetic-field regime and the available training budget. Further experiments with larger h values would be required to verify this hypothesis. Even though in $h = 2.0$ the strength of the transverse magnetic field was the highest, Figure 13d did not present the highest variability across all depth heatmaps.

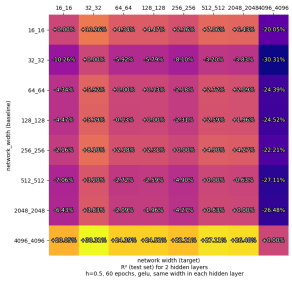
The extended Case 1 experiment is consistent with the hypothesis that larger training budgets at higher h values may recover trends similar to those observed in the other regimes. The relation between epoch budget and h value is not clear, as the low variability in Figure 13a could arise from many architectures which converge to similar solutions, and therefore decrease variability. Another possibility for low variability between different network depths is the fraction of the search space selected to create a heatmap in Figure 13a. In Figure 13a, shallow neural network architectures are clear winners, although the average performance across networks with different depths varies only by a maximum of 5%.

Overall, increasing network depth alone does not consistently improve NQS performance. The preferred depth varies across TFIM regimes, suggesting that network depth interacts with the underlying physical regime rather than acting as an independent predictor of model quality. Furthermore, the extended Case 1 experiment indicates that longer training can reduce the observed differences between depths.

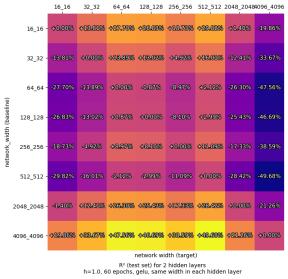
8.2.4 Network width vs. performance



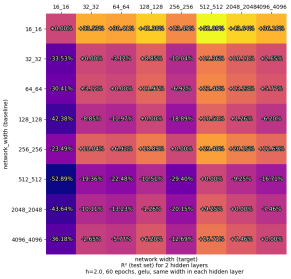
(a) Comparison on the extended Case 1 architecture dataset with $h = 2.0$, using architectures trained for 1000 epochs (81 architectures).



(b) Comparison on the full architecture dataset with $h = 0.5$, using architectures trained for 60 epochs.



(c) Comparison on the full architecture dataset with $h = 1.0$, using architectures trained for 60 epochs.



(d) Comparison on the full architecture dataset with $h = 2.0$, using architectures trained for 60 epochs.

Figure 14: Aggregated impact comparison for network widths for different regimes. Only networks with 2 hidden layers are considered, using the GeLU activation function, and each architecture has the same width in each hidden layer.

In Figure 14b, rows indicate the baseline network width, which is replaced by a target network width (columns). Based on this substitution, the average R^2 performance difference across all architectures is measured. The heatmap for $h = 0.5$ shows smaller variations between width replacements compared with the other regimes. Every heatmap contained at least one width configuration associated with substantially lower average performance (see navy-blue colors in Figure 17). The trend in an outlier network width can be interpreted as contrary across different regimes. Even though the outliers are extreme architectural width choices (such as $width = 16$ or $width = 8192$), these extremes occur at either minimum or maximum values. In Figure 14b, the network with fixed $width = 32$ in all two hidden layers achieved the best average performance across all width choices in this regime, demonstrating that the average best-performing width is neither the minimum nor maximum value. Overall, in Figure 17, the trend of choosing a reasonably well-performing network width is ambiguous - simply increasing the network width did not lead to discovering the best-performing architectures in this search space for every regime.

The similar effect as in Figure 14b was studied in Figure 14c for the $h = 1.0$ regime. Even though Figure 14c presents a similar outlier $width = 8192$ as in Figure 14b, the relation has more ambiguity than Figure 14b. In $h = 1.0$, the best-performing option is $width = 512$, which is close to the extreme outlier value of $width = 8192$ with the worst performance. Similarly, as in Figure 14b,

$width = 4096$ in Figure 14c gives the lowest average performance across all possible width choices. The interesting observation about the width performances is the lack of locality - widths with a similar number of neurons do not always perform comparably.

Figure 14d presents a similar trend in data as Figure 14c; however, the performance differences between different width choices are higher in Figure 14d than in Figure 14b. Although Figure 14d gives the same best network width as in $h = 1.0$ ($width = 512$), it gives the opposite extreme network width architectural choice as the one with the worst average performance (see $width = 16$ in Figure 14d). The heatmaps show that the width associated with the lowest performance changes between regimes. This suggests that poor-performing width choices are not universal and depend on the transverse-field regime. The main insight from Figure 14d is that replacing the minimum width value with the maximum value does not lead to the highest average change in the R^2 test set metric. Although no monotonic relationship exists between width and performance, intermediate widths (e.g., 32 or 512 depending on the regime) tend to provide competitive performance across multiple datasets. The 512 width performs reasonably well across all three regimes. The heatmap color composition is based on the highest and lowest observed difference. The occurrence of similar colors on the heatmap near different width replacements does not indicate a reasonably uniform performance across the network widths. Even though the colors for most of the replacements performed in Figure 14d look very similar, the absolute R^2 difference can exceed 0.1.

Figure 14a shows how the relation between different width architectural choices affects the "extended Case 1" architecture dataset. Even though the heatmap in Figure 14a presents the most extreme change in average R^2 performance difference for changing $width = 16$ to $width = 32$ (73.09%), the average change between two different widths in Figure 14a is not the highest among all demonstrated heatmaps (the variability in Figure 14a is not the highest among all regimes). Figure 14a demonstrates the weakest relationship between neighboring width values and performance. Even though $width = 16$ is the minimal extreme width choice, the worst-performing width choice is $width = 32$, only one step away in the explored width grid. Even though Figure 14a presents only a small fraction of the search space (81 architectures), the results could be biased based on the fraction of the search space selected. The Figure 14a search space is extended by one additional network with $width = 8192$, which performs comparably to the rest of the width choices, showing that building architectures with the highest width as possible does not lead to achieving the best average R^2 performance. Since, for the heatmap comparison, only networks with 2 hidden layers, the GeLU activation function, and the same width in each hidden layer were chosen, the above-mentioned selection decision could affect the results obtained in Figure 17.

8.2.5 Training time vs. performance

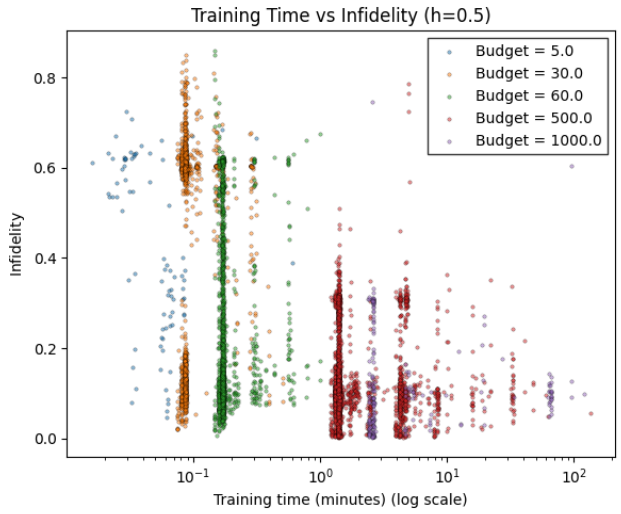
Figure 15a examines the influence of training time on the *infidelity* metric, clustered by the allocated epoch budgets. Since the architectures in the dataset were trained using different machines (ALICE high-performance computing facility (see Section 7.2.1) and a personal computer), there are architectures that, with the same epoch budget, were trained for a totally different amount of time. Overall, the clusters present for the five most frequently occurring epoch budgets present high variability in the infidelity metric, showing that high-performing architectures (architectures approaching the lowest observed infidelity values) can occasionally be discovered even with relatively small epoch budgets. As the allocated training budget increases, the observed variance in infidelity decreases, suggesting that larger epoch budgets may improve the consistency of optimization

outcomes. All four infidelity figures indicate that the epoch budget alone is not the determining factor for the total training time, since the training time can vary substantially within the same epoch budgets (see Figure 15a, where the training time for 1000 epochs can vary by orders of magnitude). One possible explanation for the variation in wall-clock time is the difference in computational complexity caused by architectural choices, such as network width and depth. However, the influence of these architectural choices on infidelity was not directly evaluated here. Therefore, wall-clock training time should not be interpreted as a direct proxy for optimization progress, since it is affected by both epoch budget and architectural complexity. It was not compared how strongly those architectural choices influence infidelity (the heatmaps show only the average differences in the R^2 metric). The observed spread of infidelity values differs between magnetic-field regimes, with maximum observed infidelity values ranging from approximately 0.4 to 0.8.

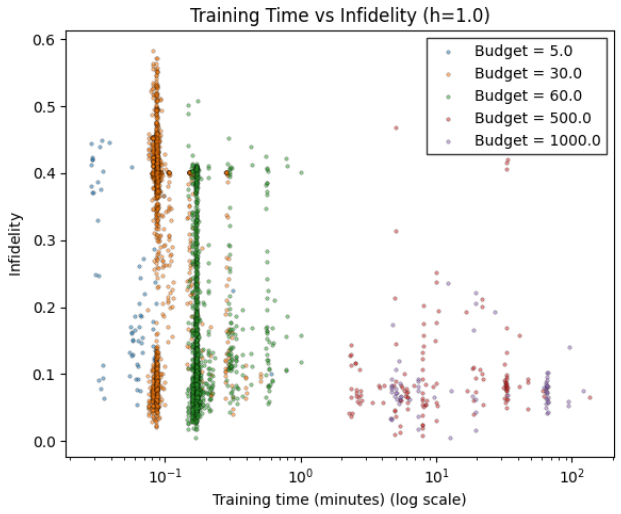
Figure 15b in $h = 1.0$ follows a similar trend as Figure 15a; however, fewer architectures are observed at longer training times, with a maximum infidelity of approximately 0.6. The clusters in Figure 15b remain separated according to training time, while the infidelity distribution within each cluster changes.

Figure 15c shows the same trend while increasing the h value as Figure 15b; however, the observed infidelity range extends to approximately 0.4, demonstrating an unexpected result. Within the sampled architectures, the $h = 2.0$ regime contains a larger fraction of low-infidelity solutions compared with the lower-field regimes, although this may also reflect differences in the sampled architecture space and optimization dynamics.

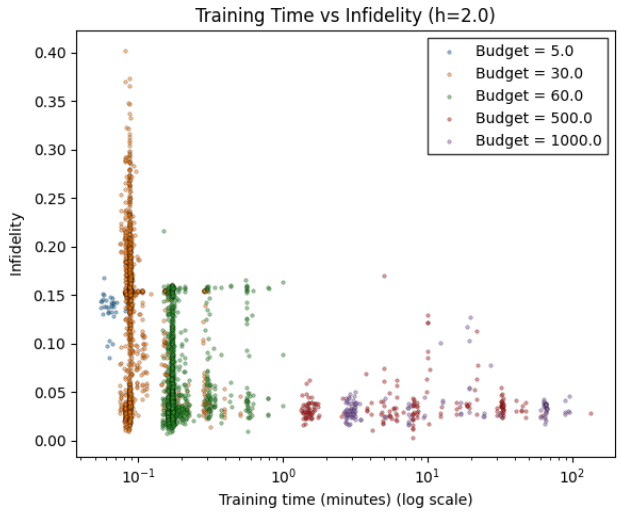
Figure 15d demonstrates how increasing the epoch budget changes the variance of the infidelity metric. Since Figures 15a, 15b, and 15c already contain architectures achieving near-zero infidelity at relatively small epoch budgets, increasing the training budget is not necessary for discovering high-performing architectures within the explored search space, although it may reduce performance variability.



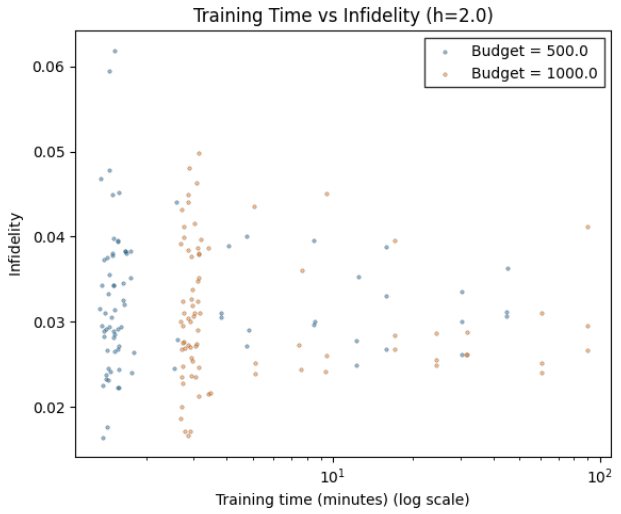
(a) Comparison on the full architecture dataset with $h = 0.5$.



(b) Comparison on the full architecture dataset with $h = 1.0$.



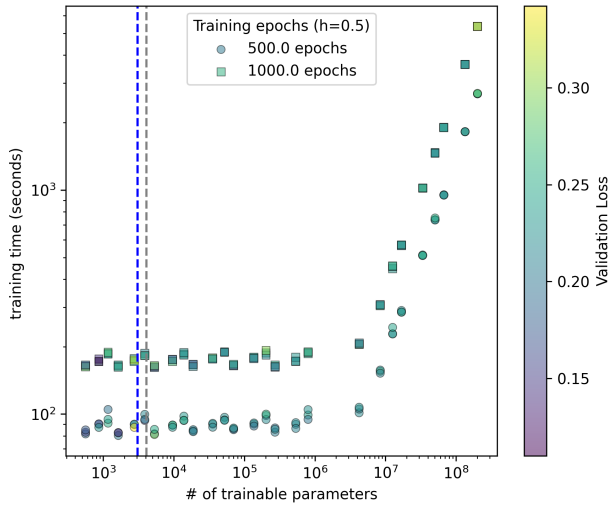
(c) Comparison on the full architecture dataset with $h = 2.0$.



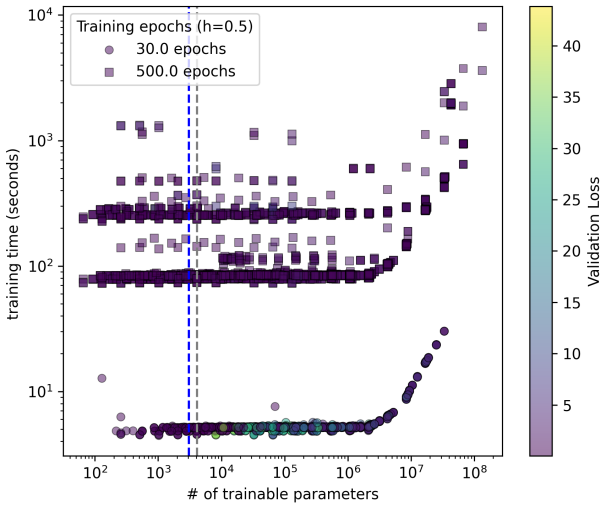
(d) Comparison on the extended Case 1 architecture dataset with $h = 2.0$.

Figure 15: Infidelity vs. training time of architectures for different regimes, colored according to the five most frequently sampled epoch budgets to highlight differences between training-budget groups.

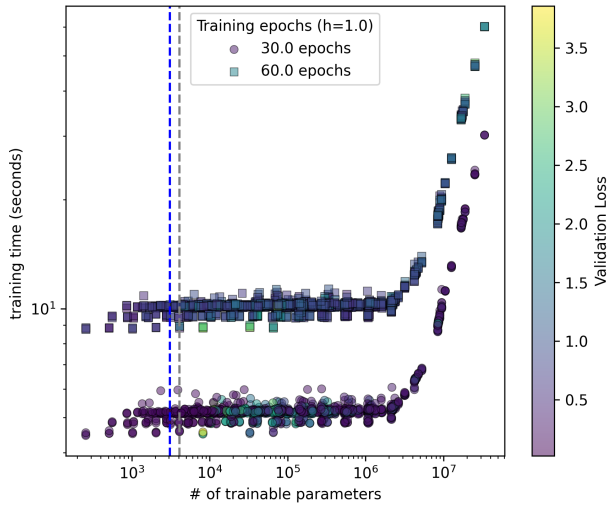
8.2.6 Trainable parameters vs. performance



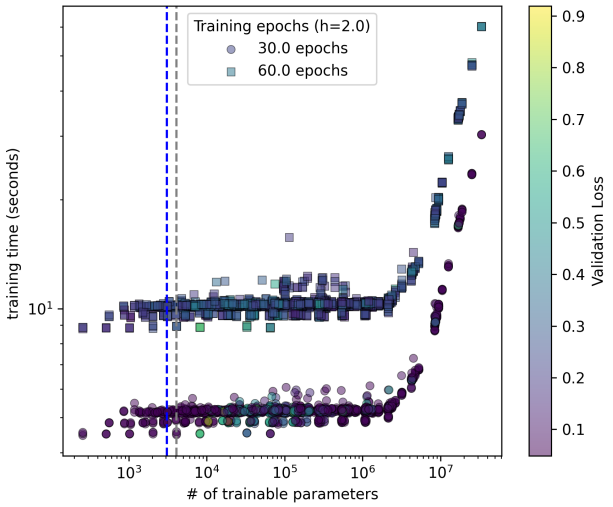
(a) Comparison on the extended Case 1 architecture dataset with $h = 2.0$.



(b) Comparison on the full architecture dataset with $h = 0.5$.



(c) Comparison on the full architecture dataset with $h = 1.0$.



(d) Comparison on the full architecture dataset with $h = 2.0$.

Figure 16: Training time vs. trainable parameters, color coded by MSE validation loss, separated using the top 2 most frequently occurring epoch budgets in each dataset. The blue dashed line indicates the point corresponding to 75% of the Hilbert-space dimension, while the gray dashed line indicates the full Hilbert-space dimension. Together, these lines provide reference points for comparing under- and overparameterized architectures.

Figure 16b investigates the relationship between the number of parameters (on a logarithmic scale), training time (on a logarithmic scale), and validation MSE loss of models in the $h = 0.5$ dataset. Figure 16b shows a positive relationship between the number of parameters and training time,

whereas the relationship between model size, training cost, and validation performance is less direct. Similarly to the infidelity metric, high-performing architectures can be discovered without requiring the largest parameter count or longest training time. This comparison provides a reference for whether the neural network parameterization is smaller or larger than the size of the represented quantum state space. The blue dashed line corresponds to 75% of the Hilbert space dimension, which represents the fraction of configurations used during training. All figures show clusters associated with epoch budgets and demonstrate that some high-performing architectures occur among relatively small models with short training times. In all figures, the best architecture is not the most computationally intensive one. Although increasing training time reduces outliers, computational resources alone do not guarantee optimal performance.

Even though Figure 16c follows a similar trend as Figure 16b, the ranges of validation loss are different, while the range of trainable parameters is the same. One hypothesis for the lack of the third cluster of architectures (see Figures 16b and 16c) is the fraction of search space trained on another machine (not ALICE). Besides changing the range of the validation loss, the range of training time also changes (in Figure 16b, training time oscillates between 10^1 to 10^4, while in Figure 16c, the training time focuses around 10^1). One possible reason is changing the epoch budget (30 and 500 epochs in Figure 16b to 30 and 60 epochs in Figure 16c).

Even though Figure 16d follows a similar trend in data as the rest of the figures, the interesting fact is that with increasing h value, the range of validation loss starts at 0.1, which is achieved using a limited training budget of 30 or 60 epochs. This result is extraordinary, as the colors present in Figure 16d and the validation loss range present the highest-performing architectures on the MSE validation loss, indicating that high-performing architectures can still be obtained in the strong transverse-field regime.

Architectural choices appear to have a stronger influence on performance than computational budget within the explored search space. The relation between the number of parameters, training time, and the Hellinger loss on the validation set is shown in Figure 16a, showing architecture performance for much higher epoch budgets. However, the minimum occurring validation loss is approximately 0.15, which is more than in Figure 16d. This indicates that increasing the training budget alone does not compensate for differences in the sampled architecture subset, and that architecture selection remains a dominant factor.

8.3 NAS algorithms

The goal of the NAS algorithms in this study is to identify architectures that achieve high predictive performance under specified computational constraints. The algorithms compared have access to the evaluation metrics of all trained architectures training results. In a real setting of finding an optimal architecture for a specified problem, the algorithms need to discover a well-performing architecture without a lookup table and exhaustive training of the entire search space. To reduce the computational cost associated with training architectures for wave-function amplitude prediction, the search is performed using precomputed evaluation metrics rather than by retraining each candidate architecture. The resulting performance comparison provides insight into which algorithms are most effective within this reproducible evaluation framework.

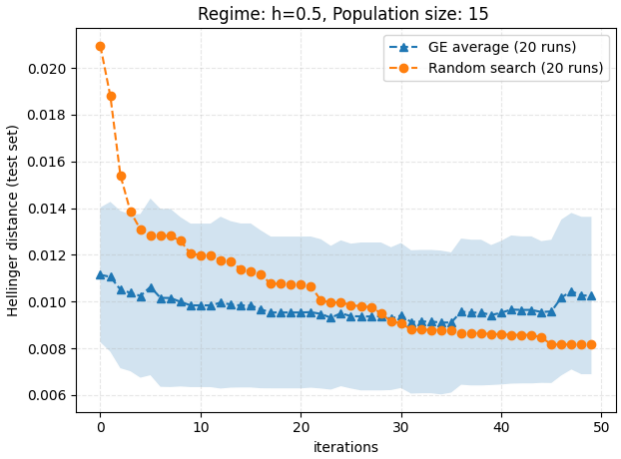
8.3.1 Evolutionary search

The general evolutionary algorithm was inspired by [56]. The algorithm maintains and iteratively updates a population of candidate architectures. In each iteration, the algorithm samples a set of parent architectures from the current population. The selected parent architectures are mutated to generate offspring (new candidate solutions), which are then added to the population. In the end, the architecture with the lowest Hellinger test loss is selected. The motivation for choosing the evolutionary algorithm is flexibility, conceptual simplicity, and competitive results [56].

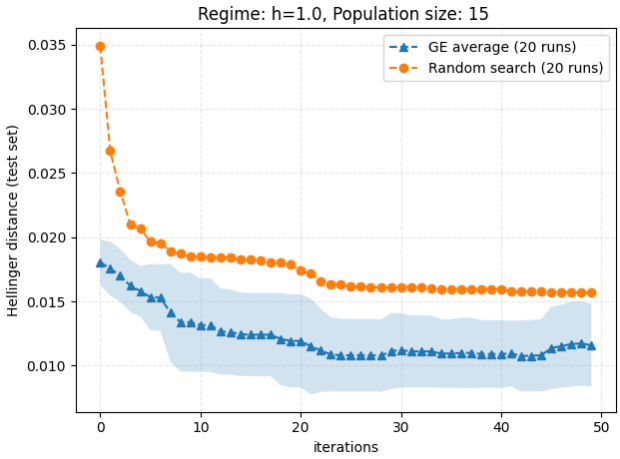
In the evolutionary search setup used in this study, in each iteration, the oldest architecture is removed from the population. The available pool for the evolutionary algorithm is the set of all architectures in a desired regime. The performance of the evolutionary algorithm depends on the choice of population size, with a positive correlation between increasing the population size and performance. In every algorithm iteration, the oldest architecture is removed from the population, and a mutated version of the best architecture is added to the population. The results of running the evolutionary algorithm are averaged across 20 runs, since NAS methods can have high variance due to the randomness of the algorithm; therefore, running multiple trials is important to verify fair comparisons. The possible mutations include activation function (selecting an activation function different from the one used by the best architecture in the population), mutating network depth (selecting a width sequence with a different number of hidden layers), or mutating network width (changing the width sequence while keeping the same number of hidden layers). The general evolutionary algorithm is compared against random search, which performs reasonably well, and a simple baseline that is fairly competitive with NAS algorithms [56].

Figure 17a shows the performance of the general evolutionary algorithm (GE) against random search (RS) for $h = 0.5$. The blue shaded region represents the standard deviation around the mean performance and shows how much the GE algorithm's performance varies from run to run. Both RS and GE improve over time, finding architectures with lower Hellinger distance on the test set as the number of iterations increases. RS improves quickly at the beginning, especially in the first 5–10 iterations. GE is more stable than RS and leads to more gradual improvement. For $h = 0.5$, random search starts with worse performance, but RS begins to outperform GE after approximately 38 iterations. The Hellinger distance on the test set ranges between 0.006 and 0.02. For $h = 0.5$, there is no clear advantage to using GE, since after 50 iterations, RS finds an architecture with higher performance.

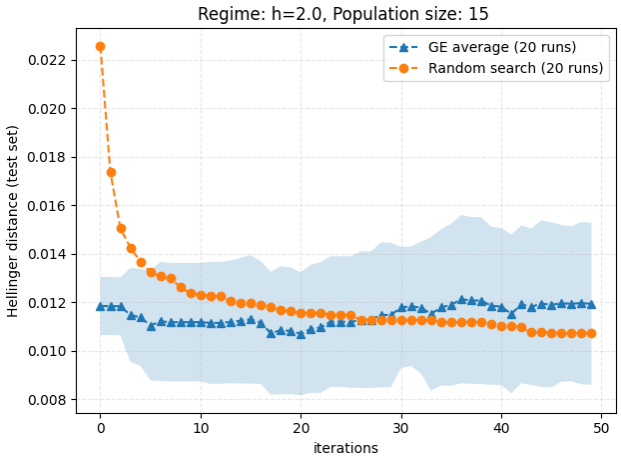
In the $h = 1.0$ regime (Figure 17b), both algorithms show similar behavior. However, in the $h = 1.0$ regime, there is no crossover between GE and RS. RS never outperforms GE, and GE finds a better-performing architecture, dominating RS throughout the search. In the $h = 2.0$ regime (Figure 17c), RS and GE behave similarly to the $h = 0.5$ regime, and RS starts to outperform GE after roughly the same number of iterations. In the extended Case 1 dataset (Figure 17d), both algorithms start to fluctuate around the 5th iteration. After roughly 16 iterations, RS finds a better architecture, leading to a substantial difference compared with GE by the 50th iteration. In Figures 17a and 17c, random search improves rapidly initially but then converges at a higher Hellinger distance. In each regime, GE appears to be more stable than RS, which can be an advantage during a small number of iterations. The blue shaded regions indicate that the standard deviation is the lowest in Figure 17b, moderate in Figures 17c and 17a, and the highest in Figure 17d.



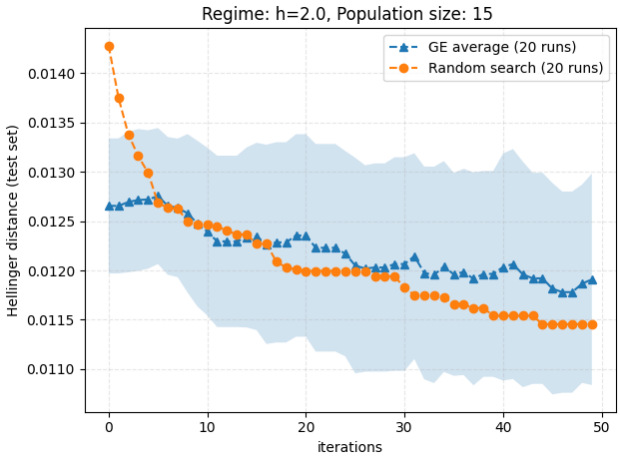
(a) Comparison on the full architecture dataset with $h = 0.5$.



(b) Comparison on the full architecture dataset with $h = 1.0$.



(c) Comparison on the full architecture dataset with $h = 2.0$.



(d) Comparison on the extended version of the Case 1 architecture dataset with $h = 2.0$ (162 architectures).

Figure 17: Performance comparison of the general evolutionary NAS algorithm and random search over 50 iterations, averaged across 20 runs under different Hellinger distance metric on the test set for different regimes.

9 Conclusions and Further Research

This thesis demonstrated the use of NQS for parameterizing quantum many-body wavefunctions. An extensive set of architectures was generated, trained, and evaluated to discover search space insights and evaluate NAS algorithms. The main contribution is developing a tabular NAS benchmark for approximating quantum many-body wavefunctions. This work gave insights into which architectural choices and NAS algorithms work best for this approach. This thesis builds upon the ideas introduced in NAS-Bench-101, which was a baseline for future study in the NAS

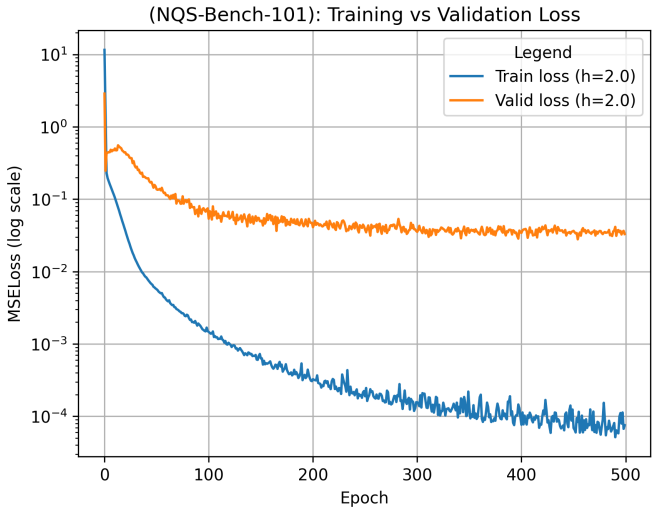


Figure 18: The MSE loss curve for the best model in regime $h = 2.0$. There is a large performance gap between training and validation curves.

domain. The developed benchmark incorporates new ideas and extensions that improve upon the efficiency of previous benchmarks. A major limitation of tabular benchmarks is the computational cost required to generate and evaluate the architectures included in the dataset. Although the Hellinger loss resulted in more stable training, MSE achieved better final performance. Overall, the dataset statistics suggest that neural architecture design has a substantial impact on wavefunction amplitude prediction performance. Although the dataset contains many poorly performing architectures and outliers that introduce strong skewness across all evaluation metrics, a significant fraction of architectures achieve accurate predictions. Across all regimes, validation metrics strongly correlate with test performance, indicating that validation-based evaluation is a reliable indicator of generalization. However, differences between training, validation, and test performance reveal signs of overfitting in some cases (see Figure 18).

The comparison between different h regimes indicates that model performance is highly dependent on the h value, with larger h values introducing greater variability and a higher number of poorly performing architectures. Increasing the training budget reduces performance variance and improves infidelity; however, computational resources alone do not guarantee better results. Instead, architecture configuration, particularly network depth and width, remains the dominant factor influencing performance. The best-performing architecture in the full dataset was not the most computationally intensive one, demonstrating that efficient architecture selection is more important than simply following a "bigger is better" approach. NAS experiments further showed that evolutionary search provides more stable improvements compared with random search, although random search can achieve competitive or even superior performance in certain regimes.

As future work, the search space could be extended with additional operations, activation functions, depths, and widths. This expanded dataset could then be used to develop a bootstrapping method, where there is a toy version of the original dataset with a much smaller number of architectures to measure general trends of the dataset (similarly as with NAS-Bench-Mini [58]). Also, a more extensive set of NAS algorithms could be developed and evaluated. An important extension would be to evaluate the proposed framework on the two-dimensional Ising model, which

exhibits richer interactions than the one-dimensional system considered in this thesis [11]. In future work, skip connections could be included in the search space. Further research could investigate alternative loss functions that combine these advantages. As in NAS-Bench-101 setting, parallel processing for training architectures could used, instead of training iteratively one architecture after another to train the entire search space for higher epoch budgets and different seeds. As future work, a Bayesian optimization (BO) algorithm could be used to optimize the hyperparameters, or a joint optimization of neural network architectures and hyperparameters could be performed. As a future study, it could be investigated whether the extremely poor-performing outliers in the architecture dataset originate from early project stages or debugging runs, where the dataset may not have been properly cleaned.

References

[1] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[2] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[3] Midhun T Augustine. A survey on universal approximation theorems. *arXiv preprint arXiv:2407.12895*, 2024.

[4] Kelly Azevedo, Luigi Quaranta, Fabio Calefato, and Marcos Kalinowski. A multivocal literature review on the benefits and limitations of automated machine learning tools. *arXiv preprint arXiv:2401.11366*, 2024.

[5] Rafael Barbudo, Sebastián Ventura, and José Raúl Romero. Eight years of automl: categorisation, review and trends. *Knowledge and Information Systems*, 65(12):5097–5149, 2023.

[6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[7] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[8] Gemma A Calvert. Crossmodal processing in the human brain: insights from functional neuroimaging studies. *Cerebral cortex*, 11(12):1110–1123, 2001.

[9] M Campbell, AJ Hoane, and Fh Hsu. Deep blue. vol. 134. *Artif. Intell*, pages 00129–1, 2002.

[10] Augustin Cauchy et al. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847.

[11] Barry A Cipra. An introduction to the ising model. *The American Mathematical Monthly*, 94(10):937–959, 1987.

[12] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[13] Anna Dawid, Julian Arnold, Borja Requena, Alexander Gresch, Marcin Płodzień, Kaelan Donatella, Kim A Nicoli, Paolo Stornati, Rouven Koch, Miriam Büttner, et al. Modern applications of machine learning in quantum sciences. *arXiv preprint arXiv:2204.04198*, 2022.

[14] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020.

[15] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[16] OS Eluyode and Dipo Theophilus Akomolafe. Comparative study of biological and artificial neural networks. *European Journal of Applied Engineering and Scientific Research*, 2(1):36–46, 2013.

[17] Damien Ernst and Arthur Louette. Introduction to reinforcement learning. *Feuerriegel, S., Hartmann, J., Janiesch, C., and Zschech, P*, pages 111–126, 2024.

[18] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. *Advances in neural information processing systems*, 28, 2015.

[19] Luca Franceschi, Michele Donini, Valerio Perrone, Aaron Klein, Cédric Archambeau, Matthias Seeger, Massimiliano Pontil, and Paolo Frasconi. Hyperparameter optimization in machine learning. *Foundations and Trends in Machine Learning*, 18(6):975–1109, 10 2025.

[20] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and David Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep feedforward networks. *Deep learning*, 1:161–217, 2016.

[22] Trevor Hastie. The elements of statistical learning: data mining, inference, and prediction, 2009.

[23] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[24] Alexandre Heuillet, Ahmad Nasser, Hichem Arioui, and Hedi Tabia. Efficient automation of neural network design: A survey on differentiable neural architecture search. *ACM Computing Surveys*, 56(11):1–36, 2024.

[25] Geoffrey E Hinton. How neural networks learn from experience. *Scientific American*, 267(3):144–151, 1992.

[26] Yoichi Hirose, Nozomu Yoshinari, and Shinichi Shirakawa. Nas-hpo-bench-ii: A benchmark dataset on joint optimization of convolutional neural network architecture and training hyperparameters. In *Asian Conference on Machine Learning*, pages 1349–1364. PMLR, 2021.

[27] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[28] Feng-Hsiung Hsu. *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton University Press, 2022.

[29] Kamal Hussain. Artificial intelligence and its applications goal. *Artificial Intelligence*, 5(01):838–841, 2018.

[30] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer, 2019.

[31] Lorraine Jacques. Teaching cs-101 at the dawn of chatgpt. *ACM Inroads*, 14(2):40–46, 2023.

[32] Shubhra Kanti Karmaker, Md Mahadi Hassan, Micah J Smith, Lei Xu, Chengxiang Zhai, and Kalyan Veeramachaneni. Automl to date and beyond: Challenges and opportunities. *Acm computing surveys (csur)*, 54(8):1–36, 2021.

[33] Christof Koch. How the computer beat the go player. *Scientific American Mind*, 27(4):20–23, 2016.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[35] Nalinda Kulathunga, Nishath Rajiv Ranasinghe, Daniel Vrinceanu, Zackary Kinsman, Lei Huang, and Yunjiao Wang. Effects of the nonlinearity in activation functions on the performance of deep learning models. *arXiv preprint arXiv:2010.07359*, 2020.

[36] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[37] Farhad Maleki, Nikesh Muthukrishnan, Katie Ovens, Caroline Reinhold, and Reza Forghani. Machine learning algorithm validation: from essentials to advanced applications and implications for regulatory certification and deployment. *Neuroimaging Clinics of North America*, 30(4):433–445, 2020.

[38] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[39] Yash Mehta, Colin White, Arber Zela, Arjun Krishnakumar, Guri Zabergja, Shakiba Moradian, Mahmoud Safari, Kaicheng Yu, and Frank Hutter. Nas-bench-suite: Nas evaluation is (now) surprisingly easy. *arXiv preprint arXiv:2201.13396*, 2022.

[40] M Schuyler Moss, Alev Orfi, Christopher Roth, Anirvan M Sengupta, Antoine Georges, Dries Sels, Anna Dawid, and Agnes Valenti. Double descent: When do neural quantum states generalize? *arXiv preprint arXiv:2508.00068*, 2025.

[41] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[42] Allen Newell, Herbert Alexander Simon, et al. *Human problem solving*, volume 104. Prentice-hall Englewood Cliffs, NJ, 1972.

[43] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[44] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[45] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach, 4/e. 2020.

[46] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[47] Rylan Schaeffer, Zachary Robertson, Akhilan Boopathy, Mikail Khona, Ila R Fiete, Andrey Gromov, and Sanmi Koyejo. Divergence at the interpolation threshold: Identifying, interpreting & ablating the sources of a deep learning puzzle. 2023.

[48] Ulrich Schollwöck. The density-matrix renormalization group. *Reviews of modern physics*, 77(1):259–315, 2005.

[49] Robert Schulz. Alice hpc wiki, 2022. Leiden University High Performance Computing Wiki (web pages).

[50] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[51] Julien Niklas Siems, Lucas Zimmer, Arber Zela, Jovita Lukasik, Margret Keuper, and Frank Hutter. Nas-bench-301 and the case for surrogate benchmarks for neural architecture search. 2020.

[52] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[53] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[54] Sean Andrew Thawe. Classical programming and machine learning.

[55] Alan M Turing. Computing machinery and intelligence. In *Parsing the Turing test: Philosophical and methodological issues in the quest for the thinking computer*, pages 23–65. Springer, 2007.

[56] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023.

[57] Chao Xue, Jiaxing Li, Xiaoxing Wang, Yibing Zhan, Junchi Yan, and Chun-Guang Li. On neural architecture search and hyperparameter optimization: A max-flow based approach. *Neural Networks*, 188:107507, 2025.

[58] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International conference on machine learning*, pages 7105–7114. PMLR, 2019.

[59] Arber Zela, Julien Siems, and Frank Hutter. Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search. *arXiv preprint arXiv:2001.10422*, 2020.

[60] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.