



國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

高效的模型架構生成基於可逆神經網路應用於神經網

路架構搜索

Efficient Neural Architecture Generation with an

Invertible Neural Network for

Neural Architecture Search

陳冠穎

Guan-Ying Chen

指導教授：周承復 博士

Advisor: Cheng-Fu Chou, Ph.D.

中華民國 112 年 7 月

July, 2023

國立臺灣大學碩士學位論文
口試委員會審定書

MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

高效的模型架構生成基於可逆神經網路應用於神經網
路架構搜索

Efficient Neural Architecture Generation with an Invertible
Neural Network for Neural Architecture Search

本論文係陳冠穎君（學號 R10922176）在國立臺灣大學資訊工程
學系完成之碩士學位論文，於民國 112 年 7 月 7 日承下列考試委員審
查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering
on 7 July 2023 have examined a Master's thesis entitled above presented by CHEN GUAN YING
(student ID: R10922176) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

周承復

(指導教授 Advisor)

呂政修

黃志輝

廖怡君

吳曉文

系主任/所長 Director:

洪士灝

Acknowledgements



首先，我要感謝我的指導教授周承復老師。在我很晚才備取台大資工所之時，他給予了我寶貴的求學建議，並且慷慨地接受我作為他的研究生。在過去的兩年碩士學習期間，周老師給了我在研究上的許多重要指導與建議，使我能順利完成碩士論文。同時，老師也教導我如何面對未知的問題，逐步分析和解決它們。

我還要感謝楊大煒學長。在我們一起進行專案的過程中，學長引導我尋找可能的研究方向，在初期為我打下了這個研究領域的基礎。他常常抽出時間與我討論研究上的困難，給予我許多幫助。學長對我完成碩士論文也功不可沒。

此外，我還要感謝我的同學們柯宏穎、吳添毅、廖盛弘、黃佳琪。無論是平時一起討論研究問題，還是在口試期間互相幫忙聆聽彼此的報告內容並提供建議，甚至在資料準備上給予幫助，你們都是我完成碩士學位考試的重要幫手。

最後，我要感謝我的家人們以及我的女友，一路上不論遇到何種挫折，他們始終陪伴在我身邊，給予我信心和支持，讓我毫無後顧之憂地完成我的碩士學位。在此，我向一路上幫助我和支持我的師長、親朋好友們致以最誠摯的謝意。


摘要



近年來，如何快速且自動化地找出預測準確度較佳的神經網路架構已備受重視。在實作上，要得到每個神經網路架構的真實預測表現是非常耗時且消耗運算資源的，因為需要在給定的資料集上實際訓練每個神經模型來取得。如何在有限的時間以及已標記好預測準確度的神經網路架構資料下，尋找出預測準確度較佳的神經網路架構是首要目標。為減少搜尋時間、運算資源，使用較少的已標記資料是優先考量的方法，因此，使用代理模型來預測每個神經網路架構準確度的方式逐漸受到採用，配合基因演算法或者最佳化演算法，例如：Local Search、Random Search、Bayesian Optimization，在預先定義好的架構搜尋空間中，搜尋出預測準確度較高的神經網路架構。然而，部分的做法只使用了已標記的訓練資料，忽略了整個搜尋空間中未標記準確度的神經網路架構資料也可以被有效利用。

本篇論文提出的方法基於可逆神經網路 (invertible neural network) 以及變分自編碼器 (variational autoencoder)，由給定的預測準確度來回推出可能的神經網路架構。此方法可有效利用整個搜尋空間中，未標記實際預測準確度的神經網路架構當作預訓練 (pre-training) 資料，利用自監督式學習 (self-supervised learning) 的技術來訓練變分自編碼器。接著，我們能利用變分自編碼器中的 Encoder 來將神經網路架構，由離散的空間轉換到連續的平坦空間，再使用可逆神經網路做回歸建模任務 (regression) 的訓練，利用神經網路模型架構的平坦空間表示 (latent representation) 預測出該模型架構的真實準確度。最後，利用可逆神經網路的特性，我們能夠逆推出表現較好的模型架構，並且搭配本方法也有代理模型的特性，可預測出神經網路架構的準確度，挑選出可能的候選架構，經過每一輪的逆推與再訓練疊代後，我們的模型最終能回推出表現較佳的神經網路架構，達到找尋出準確度較高的神經網路架構的目標。

在實驗中，我們將提出的方法做效能評估，利用 Neural Architecture Search (NAS)



領域常用來比較的公開的神經網路架構搜尋評估庫(benchmarks)與其他方法比較，這些公開的評估庫讓 NAS 的研究有一個可以公平比較的平台，包含：NAS-Bench-101、NAS-Bench-201，根據實驗結果，我們提出的做法可以在有限的已標記資料下達到很好的表現，能夠搜尋出表現較高的神經網路架構，在與相同領域論文的實驗結果比較後，展現出我們的方法與當今最先進的做法(state-of-the-art)是可以相比擬的。


關鍵詞：神經網路架構搜索、機器學習、圖神經網路、變分自編碼器、生成式模型、可逆神經網路

Abstract



In recent years, there has been an increasing fascination with the efficient and automated discovery of high-performing neural architectures. However, evaluating performance of each architecture is time-consuming as it requires actual training on a prepared dataset. Therefore, the primary goal is to search for well-performing neural architectures within a limited set of architectures that have been evaluated. To reduce the need for actual training and labeled data, using surrogate models to predict the performance of neural architectures has become popular. This approach is often coupled with genetic algorithms or optimization algorithms such as Local Search (LS), Random Search (RS) and Bayesian Optimization (BO) to identify better neural architectures within the predefined search space. However, it has been observed that some methods only use labeled training data and do not make full use of available unlabeled data, i.e., all untrained architectures themselves in the search space.

Our method is based on Invertible Neural Network (INN) to inversely map the neural architecture from its performance. This method makes full use of the unlabeled data (untrained neural architectures) within the entire search space to train a variational autoencoder with a self-supervised learning mechanism. The variational autoencoder transforms the architecture into a latent space. Then, the invertible neural network



performs as a regressor to convert the latent representation of the architecture into its performance. Finally, the invertible neural network can be used to infer the latent representation of best-performing architectures. Coupled with the surrogate model property of our method, it can predict the performance of candidate architectures and add them to training data. Our model can iteratively learn to infer and inverse to better-performing neural architectures.

Our method is evaluated on publicly widely used benchmarks for NAS which help us to compare our work with other approaches, including NAS-Bench-101, NAS-Bench-201. The results demonstrate that our method can search for better-performing neural architectures with limited evaluated architectures and comparable with the state-of-the-art approaches.

Keywords: Neural Architecture Search, Machine Learning, Graph Neural Network, Variational Autoencoder, Generative Model, Invertible Neural Network

Contents

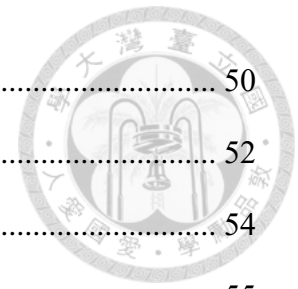


Acknowledgements	ii
摘要	iii
Abstract	v
Chapter 1 Introduction	1
Chapter 2 Related Work	7
A. Neural Architecture Search.....	7
B. NAS Benchmarks	8
1. Introduction of NAS-Bench-101	9
2. Introduction of NAS-Bench-201	9
C. Performance Predictors.....	10
D. Variational Autoencoders.....	11
E. Graph Neural Networks.....	12
1. Graph Convolutional Networks.....	13
2. Graph Isomorphism Networks.....	14
F. Graph Variational Autoencoders.....	15
G. Invertible Neural Networks	15
Chapter 3 Method	19
A. Data Preprocessing	19
1. Architecture Encoding.....	19
2. NAS-Bench-101	20
3. NAS-Bench-201	21
B. Model Architecture	22
1. Encoder.....	22
2. Decoder.....	24

3.	Invertible Neural Network.....	25
C.	Training Method	27
1.	Objective function	27
(a)	Graph Variational Autoencoder	27
(b)	Invertible Neural Network.....	28
2.	Pre-train GVAE.....	29
3.	Fine-tune INN.....	30
D.	Retrain and Search.....	31
1.	Algorithm	31
2.	Rank-based Weighted Loss.....	33
Chapter 4	Experiments	37
A.	Evaluation Metrics.....	37
1.	Architecture Search	37
2.	Regression	38
3.	Inversion.....	38
B.	NAS-Bench-101	39
1.	Architecture Search	39
2.	Regression	40
3.	Inversion.....	40
C.	NAS-Bench-201	43
1.	Architecture Search	43
2.	Regression	44
3.	Inversion.....	45
Chapter 5	Ablation Studies	50
A.	Choice of Candidates Generative Methods	50



B. Choice of Fine-tune Methods	50
C. Whether Using Rank-based Weighted Loss	52
Chapter 6 Conclusion	54
References	55
Appendices	58
A. NAS-Bench-201	58
1. Architecture Search	58
2. Regression	59
3. Inversion	62
B. Hyperparameters.....	65



List of Figures



Figure 1: Neural architecture representation of NAS-Bench-101.....	20
Figure 2: Neural architecture representation of NAS-Bench-201.....	21
Figure 3: The overview of proposed model.....	22
Figure 4: The forward and inverse processes of INN.....	26
Figure 5: GVAE.....	28
Figure 6: This visualization, created via t-SNE, demonstrates the retraining and searching process for each iteration on CIFAR-10 of the NAS-Bench-201 search space. The 100 candidates, generated by inversely mapping 1.0, are color-coded, while the non-selected neural architectures are displayed in grey.....	33
Figure 7: Regression results on NAS-Bench-101.....	41
Figure 8: Inversion results on NAS-Bench-101.....	42
Figure 9: Inversion result over a range on NAS-Bench-101.....	43
Figure 10: Inversion result over a range on CIFAR-10.....	45
Figure 11: Regression results on CIFAR-10.....	47
Figure 12: Inversion results on CIFAR-10.....	48
Figure 13: The comparison of search curves for fine-tuning methods with rand-based weighted loss on CIFAR-10 dataset of NAS-Bench-201.....	52
Figure 14: Comparison of search curves with and without the use of rank-based weighted loss on NAS-Bench-201.....	53
Figure 15: Regression results on CIFAR-100 and ImageNet16-120.....	61
Figure 16: Inversion results on CIFAR-100 and ImageNet16-120.....	63
Figure 17: Inversion results over a range.....	64

List of Tables




Table 1: The comparison results on NAS-Bench-101. The means and standard deviations are reported.	39
Table 2: The comparison results on NAS-Bench-201. The means are reported.....	44
Table 3: The comparison of the candidate generative methods on NAS-Bench-101....	50
Table 4: The comparison of the candidate generative methods on NAS-Bench-201....	50
Table 5: This table compares the two fine-tuning methods. The methods denoted with the postfix 'R' utilize a rank-based weighted loss.....	51
Table 6: A comparison of the use of rank-based weighted loss. Methods with a 'R' postfix denote the utilization of rank-based weighted loss.....	52
Table 7: The comparison results on NAS-Bench-201. The mean and standard deviation are reported.	58
Table 8: Hyperparameters of the GIN encoder.	65
Table 9: Hyperparameters of the Transformer decoder.	65
Table 10: Hyperparamters of the pretraining.....	65
Table 11: Hyperparameters of Invertible Neural Network.	65
Table 12: Hyperparameters of fine-tunning.....	65
Table 13: Hyperparameters of retraining.....	66
Table 14: Hyperparameters of searching.....	66

Chapter 1 Introduction



In recent years, there has been an increasing fascination with Neural Architecture Search (NAS) due to its ability to enable the automatic discovery of high-performing neural architectures. This approach can save significant time and effort compared to traditional manual design methods. The neural architecture search problem can be transformed into a graph optimization problem, where the goal is to find the optimal architecture represented as a directed acyclic graph (DAG). The graph's nodes symbolize various operations, including convolution and pooling, while the edges illustrate the data flow between these operations. The research on NAS aims to find the best neural architecture that can achieve good performance on a given task while minimizing the computational cost.

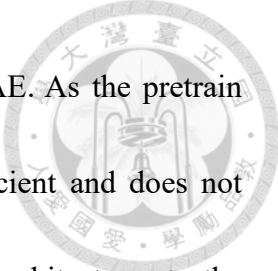
The methods based on performance predictor have recently become mature, which roughly includes [1-4]. This method combines performance predictors with search algorithms such as Random Search (RS), Local Search (LS), Evolutionary Algorithm (EA), Bayesian Optimization (BO), etc., to find the well-performing architectures. When more standard-trained architectures are available as training data, the performance predictor performs better. However, collecting labeled data means that we need to train and evaluate the architecture, and this process is extremely time-consuming.



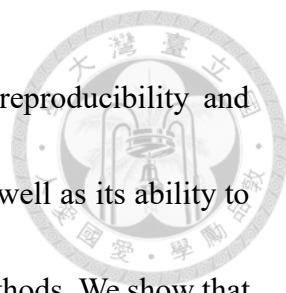
Reducing the requirement for evaluated architectures, i.e., labeled architecture data, is the key to predictor-based NAS methods. Research has been focusing on how to achieve better results with reduced labeled data, including semi-supervised learning [5] and ensemble learning [3], among others. Some works have been effective in make full use of untrained architectures within the search space as training data via self-supervised learning. This involves converting discrete architectures into a latent space, which further reduces the need for labeled data. Recent research [4, 6, 7] not only utilize the entire unlabeled data in the search space but has also focus on the latent space optimization (LSO) to obtain robust and better results.

Generative models are currently very popular for computer vision (CV) and natural language processing (NLP). Some studies in the field of NAS have also used generative methods to generate high-performing neural architectures. For example, GA-NAS [8] leveraged the concept of GANs and improved upon it, while AG-Net [9] utilized the GNNs decoder from SVGe [6] and made certain modifications to use it as a standalone generative model to iteratively generate better neural architectures. This type of approach often makes full use of unlabeled architecture within the search space, resulting in better performance.

In our approach, we proposed a method that combines a graph variational autoencoder (GVAE) with an invertible neural networks (INNs) [10]. We utilized the

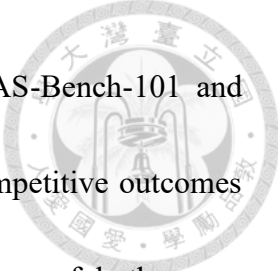


entire set of unlabeled data in the search space to pretrain the GVAE. As the pretrain process does not require accuracy label for architectures, it is efficient and does not require any queries on tabular benchmarks. The encoder maps the architectures to the latent space and we fine-tune the INNs with the latent representation and the accuracy of each architecture as the objective. The INNs can be used to inverse the accuracy to the latent representation, and then the decoder can be used to decode the latent representation to obtain the corresponding neural architectures. We also can use whole model as a performance predictor to select the top architectures. Inspired by AG-Net [9], where the generator is iteratively retrained with successful architectures generated in previous iterations to gradually move towards the high-performing region, we generate the potential architectures by inverting the highest accuracy score (i.e., 1.0) and predict their performance. We then select the top-performing architectures to evaluate their true accuracy and add them to training set to retrain the model. This iterative process enables our approach to continuously improve its performance by discovering well-performing architectures and retraining the model accordingly. Our method does not rely on traditional NAS algorithms such as LS, BO, or EA, which makes it more efficient compared to those methods. Additionally, our approach generates architectures using a reliable latent representation instead of random noise, which distinguishes it from AG-Net [9], where the neural architecture is generated from a normal distribution.



Extensive experiments have been conducted to evaluate the reproducibility and stability of our proposed method with multiple independent runs, as well as its ability to improve query efficiency and search results compared to baseline methods. We show that our method is better than many existing NAS methods, including baseline methods such as RS, LS, BO, EA. Additionally, our approach exhibits competitive performance compared to state-of-the-art approaches such as BANANAS [11], AG-Net [9] and CR-LSO [7] on representative NAS benchmark sets, including NAS-Bench-101 [12] and NAS-Bench-201 [13]. We provide more detail about experiments in Chapter 4. Our contributions can be summarized as follows.

1. We make full use of unlabeled architectures within the entire search space to pre-train a graph variational autoencoder (GVAE) through self-supervised learning. This process involves embedding the discrete architectures into the latent space. By doing so, the graph encoder can extract the similarity of neural architectures and reduce the required amount of labeled data.
2. Our approach generates neural architectures by inverting their performance into a latent representation, and then employing a graph decoder to transform them into discrete architectures. This invertible property enables us to not only generate the best-performing neural architecture but also to discover one that is most proximate to a predetermined performance goal.

- 
3. Experiment results on two NAS benchmarks, specifically NAS-Bench-101 and NAS-Bench-201, demonstrate that our NAS method attains competitive outcomes in comparison to other state-of-the-art NAS methods, in terms of both query efficiency and model performance.

The remainder is organized as follows: Chapter 2 introduces the related works relevant to this paper. Chapter 3 presents the proposed method in details, including the data preprocessing, model architectures, and training pipelines. In Chapter 4, experimental settings and results are reported. Chapter 5 features an ablation study discussing the selection of fine-tuning methods, the utilization of rank-based weighted loss, and the resulting impact on performance. Lastly, Chapter 6 summarize our work and outlines directions for future research.



Chapter 2 Related Work




A. Neural Architecture Search

Essentially, Neural Architecture Search (NAS) can be transformed into a graph optimization problem that aims to find the optimal configuration of operations for each node, such as convolution, pooling, non-operation, and skip connection, within a discrete search space. Typically, these architectures are represented as directed acyclic graphs (DAGs), and we impose certain constraints to define the search space. The goal is to efficiently explore this space to identify the optimal architecture that maximizes performance while satisfying the given constraints.

The discrete nature of neural architectures poses a challenge for applying gradient-based optimization methods. To overcome this problem, some NAS approaches utilize discrete encoding schemes to representing search spaces, such as adjacency matrices and one-hot vectors of operations as node features. Additionally, alternative optimization algorithms, including Random Search (RS), Local Search (LS), Evolutionary Algorithm (EA), Bayesian Optimization (BO) have been explored in previous studies.

Recently, some neural architecture search methods have moved away from traditional discrete optimization methods and adopted faster weight-sharing approaches. This trend has resulted in the adoption of differentiable optimization techniques. For instance, DARTS (Differentiable Architecture Search) method [14] has introduced a



differentiable approach to neural architecture search problems. It involves relaxing the discrete architecture search space by modifying operator selection to differentiable operations, enabling gradient-based NAS. Following this relaxation, the NAS task evolves into the joint optimization of model architectures and weights.

Another approach is that map the discrete architecture space into a continuous latent space, enabling the search for well-performing architectures within this continuous space. In our approach, we also train a deep variational autoencoder to transform the search space from discrete to continuous. In this case, the discrete neural architectures can be searched from the continuous latent space.

B. NAS Benchmarks

To support benchmarking and evaluation for NAS works, several NAS benchmarks and well-defined search spaces have been introduced. These include NAS-Bench-101 and NAS-Bench-201, which are tabular benchmarks that enable direct querying of various metrics, such as parameters, training time, and architecture performance. These benchmarks play a crucial role in evaluating our method. Furthermore, there are surrogate benchmarks like NAS-Bench-301 and NAS-Bench-x11, which employ surrogate models to predict performance in the DARTS search space. Due to the high computational complexity and the extensive number of architectures (10^{18}) in the DARTS search space, it is infeasible to evaluate all the architectures directly. In our study, we will provide a

comprehensive introduction to NAS-Bench-101 and NAS-Bench-201, which serve as the evaluation frameworks for our method.




1. Introduction of NAS-Bench-101

NAS-Bench-101 is a comprehensive, publicly NAS benchmark containing the performance metrics of over 423K unique convolutional neural network (CNN) architectures. The dataset is generated by exhaustively training and evaluating these neural architectures on the CIFAR-10 dataset. NAS-Bench-101 aims to provide a standardized platform for researchers to conduct fair comparisons of different NAS algorithms and understand their strengths and weaknesses. This benchmark employs an operation-on-nodes mechanism. The space of a cell consists of all possible DAGs on V nodes, where the first and last nodes are labeled as operation “INPUT” and “OUTPUT” respectively, representing the input and output tensors of the cell. Each node has one of L labels. The space of labeled DAGs will grow exponentially in both V and L . In order to constraint the number of possible neural architectures within this search space, the following constraints are imposed: there are three types of operations (1x1 convolution, 3x3 convolution, 3x3 max-pool), the number of nodes is at most seven, and the maximum number of edges is nine.

2. Introduction of NAS-Bench-201

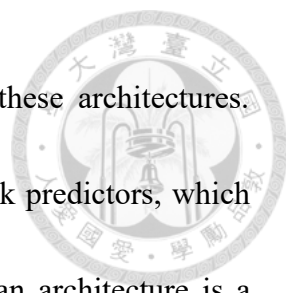
NAS-Bench-201 is a tabular NAS benchmark consists of 15,625 neural architectures.



These neural architectures have been trained and evaluated using the same setup on three diverse image classification tasks: CIFAR-10, CIFAR-100, and ImageNet-16-120. In NAS-Bench-201, the architectures are represented using cells as the basic building blocks, similar to NAS-Bench-101. By stacking these cells in a predefined order, various CNN architectures can be generated. The key difference, however, is that NAS-Bench-201 utilizes an operation-on-edges mechanism. Each neural architecture cell is represented as a directed acyclic graph (DAG), with four nodes and six edges. The predefined set contains five operations, including 1x1 and 3x3 convolutions, 3x3 avg-pooling, zero operation, and skip connections. Additionally, it provides training time and meta information for each epoch of each architecture, which can be valuable for NAS researchers. The authors also benchmarked ten NAS algorithms to analyze this search space, establishing it as a baseline. Consequently, in our work, we make some adjustments to transform the DAG into an operation-on-nodes representation for convenience.

C. Performance Predictors

An architecture performance predictor is employed to predict the performance of architectures before actually training them. The predictor can be simply formulated as a mapping function $f: X \rightarrow P$, where X represents a search space of architectures and P denotes a set of performance metrics. This technique was first applied by PNAS [15] and [16], which used a LSTM-based predictors as a surrogate model capable of handling



variable-sized sequential architectures to predict the accuracy of these architectures. WeakNAS [3] propose ensemble models consisting of multiple weak predictors, which can be trained with fewer labeled architectures. As the nature of an architecture is a computational graph, recent works [1, 2] predict the performance of architecture by graph neural networks (GNNs) encoders to extract the features from graphs. Some works [4-6] learn neural architecture representations through a variational graph autoencoder via self-supervised learning. Notably, Arch2Vec [4] employs multiple Graph Isomorphism Network (GIN) [17] layers as an encoder coupled with a simple multiplayer perceptron as a graph decoder. Our work adapts the graph encoder from Arch2Vec. These GNN-based models, which use the adjacency matrix and operation one-hot vector to represent a graph as input data, improves upon previous approaches. The results show that the highly informed latent representations are crucial for downstream performance prediction. Consequently, studies [4, 6, 18, 19] are working to find a better latent space through self-supervised approaches to enhance the NAS performances.

D. Variational Autoencoders

Variational Autoencoders (VAEs) have become a popular deep generative modeling framework since their introduction by Kingma and Welling [20]. They consist of an encoder $p_{\theta}(z|D)$ that maps high-dimensional data D (from a source domain of an unknown distribution) to a latent variable z from lower-dimensional continuous latent



space, and a probabilistic decoder $q_\phi(D|z)$ that reconstructs the original data from the latent variable z . The parameters θ and ϕ of this network are learned jointly by optimizing the evidence lower bound (ELBO) on the data likelihood:

$$L(\theta, \phi, D) = \mathbb{E}_{z \sim p_\theta(z|D)} [\ln q_\phi(D|z)] - KL(p_\theta(z|D) || q(z)). \quad (1)$$

Here, $\mathbb{E}_{z \sim p_\theta(z|D)} [\ln q_\phi(D|z)]$ represents the expected log-likelihood of the data D given the latent variable z . On the other hand, it measures the reconstruction loss, which aims to minimize the dissimilarity between the input data and the reconstructed data. $KL(p_\theta(z|D) || q(z))$ corresponds to the Kullback-Leibler (KL) divergence, a regularization term that estimates the difference between two distributions and help to regularize the latent space. VAEs have been applied to various domains, and are often used for generation tasks by taking only the decoder part.

E. Graph Neural Networks

Graph Neural Networks (GNNs) have become as a powerful framework for representation learning on graph-structured data, addressing the limitations of traditional neural networks when dealing with irregular data. GNNs were first introduced by Scarselli [21], and since then, a variety of GNN architectures have been proposed, including Graph Convolutional Networks (GCNs) [22], Graph Attention Networks (GATs) [23], and Graph Isomorphism Networks (GINs) [17].

The extraction of graph features in Graph Neural Networks (GNNs) typically



involves two steps. Firstly, GNNs learn a representation for each node using message-passing layers. These layers iteratively aggregate information from neighboring nodes, allowing the nodes to exchange information and update their features. This process allows information to propagate across the graph, enabling the GNN capture both local and global graph features. After the message-passing phase, the GNN proceeds to the second step, where a function is applied to summarize the node features. This step aims to obtain graph-level features or maintain node-level features for downstream tasks, such as node classification tasks and link prediction tasks.

For Neural Architecture Search (NAS) problems, GNNs have been utilized for architecture performance prediction, leveraging the graph-structured nature of neural architectures to enable efficient exploration of the search space..

1. Graph Convolutional Networks

Given a graph $G = (V, E)$, for each node $v_i \in V$ is associated with a feature $x_i \in \mathbb{R}^D$. Here, V denotes the set of nodes in G , and D denotes the dimensionality of the node features.

Graph Convolutional Networks (GCNs) can be formulated as follows:

$$H^{l+1} = f(H^l, A), \quad (2)$$

where $H^0 = X \in \mathbb{R}^{|V| \times D}$ is the input for the first layer, and A is the adjacency matrix.

In the formulation, H^l represents the node features at the l -th layer, and the function f



takes both the adjacency matrix and node features into account to update the node features for the subsequent layer.

A specific example of the function f is given by:

$$H^{l+1} = f(H^l, A) = \sigma(AH^lW^l). \quad (3)$$

In this equation, $\sigma(\cdot)$ represents a non-linear activation function, and W^l denotes the model weights of l -th layer.

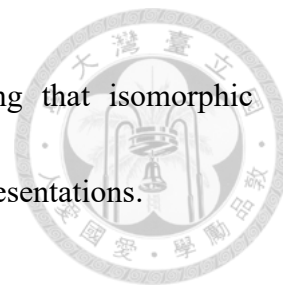
2. Graph Isomorphism Networks

Graph Isomorphism Networks (GINs) are a type of Graph Convolutional Network (GCN) introduced by [17] to address the issue of graph isomorphism, which is the problem of determining whether two graphs have the same structure despite possible differences in the labeling of nodes and edges. GIN is designed to be more powerful than some existing GCN models when it comes to distinguishing non-isomorphic graphs.

The key feature of GIN is its aggregation function, which combines node features and edge information in a manner that is sensitive to the graph structure. This allows GIN to learn expressive node embeddings and capture complex graph structures more effectively compared to some other GCN models.

In Neural Architecture Search (NAS) problems, the search space often contains many isomorphic graph architectures. These isomorphic architectures should be mapped to the same latent representation or yield identical performance. To address this challenge,

we utilize GIN layers to construct our encoder model, ensuring that isomorphic architectures are effectively handled and mapped to appropriate representations.

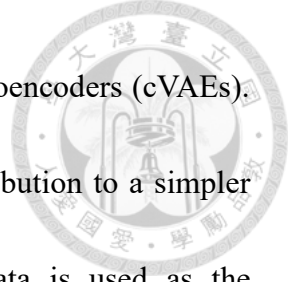


F. Graph Variational Autoencoders

Graph Variational Autoencoders (GVAEs) extend the VAE framework to handle graph-structured data, making them particularly suitable for tasks involving networks, molecular graphs, or other relational data. The GVAEs employ graph neural networks (GNNs) as encoders, taking advantage of their ability to effectively capture both local and global graph structures. GVAEs have been applied to several graph-level tasks, including graph generation, graph clustering, edge prediction, and graph classification. In our work, we use the asymmetric GVAE, where the encoder and the decoder in our GVAE model are not identical. Specifically, we use a GNN encoder to extract the features of neural architectures represented as graphs and utilize a non-GNN decoder to generate the architectures by predicting the types of nodes and whether the edges exist using the latent representation obtained from the invertible neural network.

G. Invertible Neural Networks

Invertible Neural Networks (INNs) possess a key property of bijectivity, implying that the mapping from inputs to outputs is reversible. This property allows the output to be inverted back to the input. The paper [10] significantly contributes to the understanding and application of INNs, showcasing their effectiveness in solving inverse problems. The



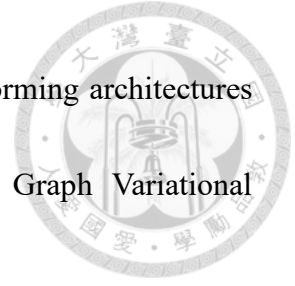
approach bears conceptual similarities to conditional variational autoencoders (cVAEs).

INNs learn a non-linear transformation that connects the data distribution to a simpler prior distribution, such as Gaussian Distribution. The output data is used as the conditioning variable in this transformation process. This setup enables the INN to generate the original input data by using the output data as a condition, coupled with a sampled prior distribution.

The training process of an INN involves learning two models: $g(z, y; \theta)$ represents the inverse process, and $f(x; \theta)$, which approximating the known forward process $s(x)$. These models are jointly trained with the objective of capturing the correlation between the input data x and the output data y by optimizing the parameters θ . Once the models are trained, the input data x can be recovered by applying the inverse model $g(z, y; \theta)$ to a sample z drawn from a standard normal distribution $\mathcal{N}(0, I_k)$, where k is the dimensionality of the input space. In another representation, $x = g(z, y; \theta)$, and the pair $[z, y]$ is obtained by applying the forward model $f(x; \theta)$ to x , such that $[z, y] = f(x; \theta) = [f_z(x; \theta), f_y(x; \theta)] = g^{-1}(x; \theta)$. Here, $f_y(x; \theta)$ approximates $s(x)$, the known forward process.

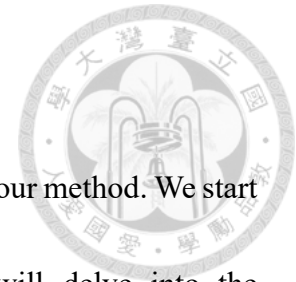
In our work, we train an INN-based performance predictor for neural architectures, treating it as a regression task. By leveraging the invertible property of the INNs, we can retrieve the latent representations of high-performing neural architectures by inversely

mapping the highest accuracy, which is set to 1.0. These well-performing architectures are obtained by decoding the latent representations using our Graph Variational Autoencoder (GVAE) decoder.





Chapter 3 Method



In this chapter, we will present a comprehensive introduction to our method. We start by discussing the preprocessing steps for the data. Next, we will delve into the architecture of our model, which includes the encoder and decoder components of the graph variational autoencoder (GVAE), as well as the setting of the invertible neural network (INN). Finally, we will explain the objective function employed to train the model and provide specific details regarding the training process.

A. Data Preprocessing

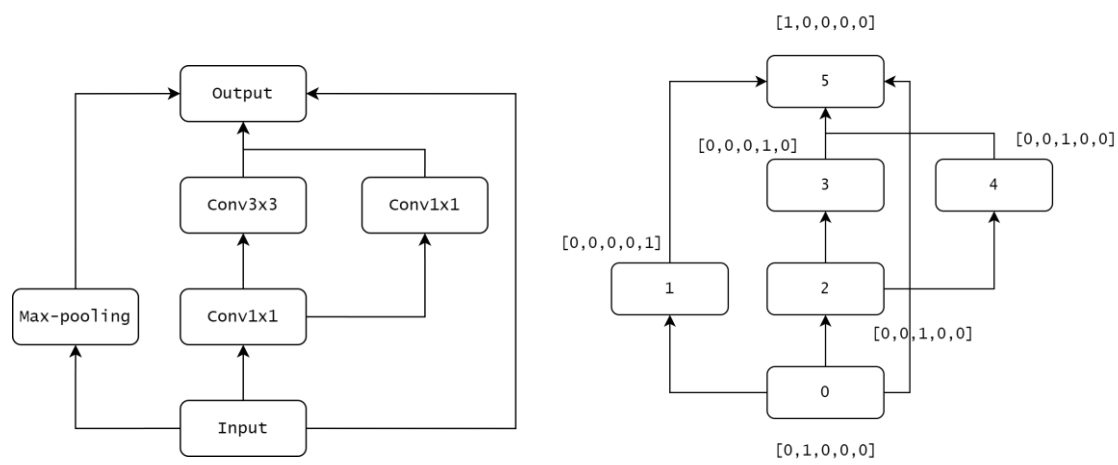
1. Architecture Encoding

Due to the nature of architectures being represented as graphs, we can utilize an adjacency matrix to indicate the connections between nodes and use a one-hot vector to represent the operation types on each node. As shown in Figure 1, (a) represents the original architecture as a graph, (b) demonstrates how each operation can be encoded using a one-hot representation, and (c) represents the adjacency matrix of the graph.

To fully leverage the features of graph convolutional networks, which aggregate the node features of each node's neighbors, we modify the adjacency matrix by taking the element-wise maximum (denoted by \vee) of the original adjacency matrix A and its transpose A^T . This operation creates an undirected graph, allowing for bidirectional information flow during graph convolutional operations. By considering both the



connections from node i to its neighbors and the connections from its neighbors back to node i , we capture a more comprehensive understanding of the graph's local and global structures. This modification enhances the effectiveness of graph convolutional networks in capturing and propagating information across the graph.



(a) Architecture representation

(b) One-hot vector

0	1	1	0	0	1
0	0	0	0	0	1
0	0	0	1	1	0
0	0	0	0	0	1
0	0	0	0	0	1
0	0	0	0	0	0

(c) Adjacency matrix

Figure 1: Neural architecture representation of NAS-Bench-101.

2. NAS-Bench-101

In the NAS-Bench-101 search space, there are five types of operations: OUTPUT, INPUT, 1×1 convolution, 3×3 convolution, and 3×3 max-pool. The number of nodes in this search space ranges from three to seven. To accommodate architectures with fewer than seven nodes, we set the adjacency matrix A as a 7×7 matrix and pad it



with zeros if the architecture has fewer than seven nodes. Similarly, we pad the operation one-hot vector with zeros to ensure consistency in dimensions.

3. NAS-Bench-201

In the NAS-Bench-201 search space, the architecture graphs have a fixed structure with four nodes and six edges. Each edge represents a specific operation. There are seven types of operations: OUTPUT, INPUT, 1×1 convolution, 3×3 convolution, 3×3 avg-pool, skip-connect, and zeroize (none). The original architecture representation is depicted in Figure 2(a). To simplify the representation, we transform it into an operation-on-node graph, as shown in Figure 2(b), resulting in a total of eight nodes and ten edges in the graph. The skeleton of the NAS-Bench-201 search space is fixed, so our focus is solely on the operations performed on the nodes.

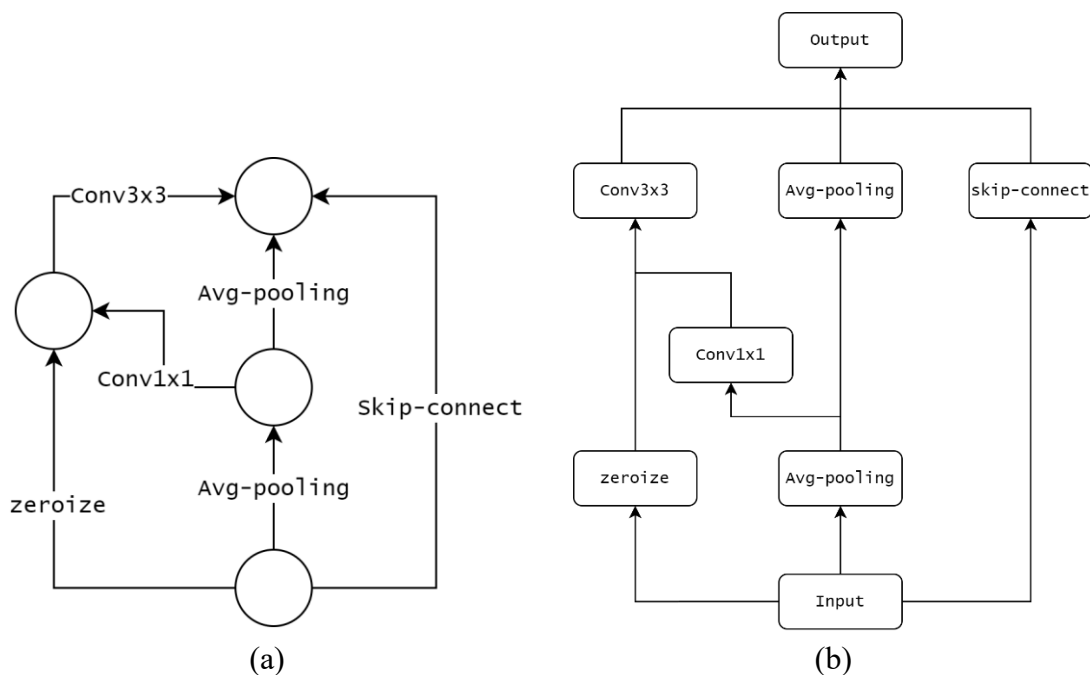


Figure 2: Neural architecture representation of NAS-Bench-201.



B. Model Architecture

We provide an overview of our model architecture in Figure 3. The encoder is responsible for transforming the graph into the latent space, while the decoder is tasked with reconstructing the graph. The INN is trained as a regression task, with the label being the accuracy corresponding to the architecture graph.

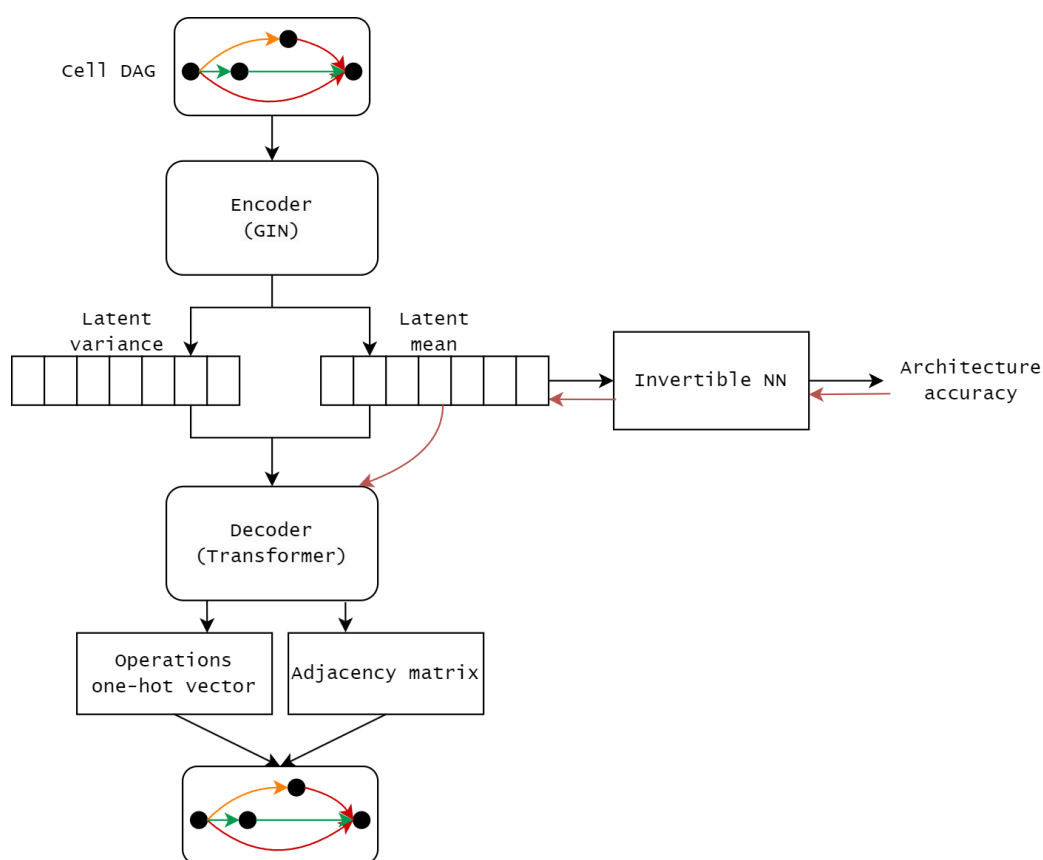
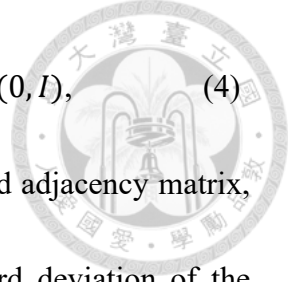


Figure 3: The overview of proposed model.

1. Encoder

To address the presence of isomorphic graphs, which often arise in neural architecture search problems, we utilize the Graph Isomorphism Networks (GINs) to construct the encoder. The variational encoder model can be defined as follows:



$$p(Z|X, \tilde{A}; \theta) = \mu(X, \tilde{A}; \theta) + \alpha \times \sigma(X, \tilde{A}; \theta) \odot \varepsilon, \varepsilon \sim \mathcal{N}(0, I), \quad (4)$$

where X, \tilde{A} represent the matrix of node features and the augmented adjacency matrix, respectively. $\mu(X, \tilde{A}; \theta)$ and $\sigma(X, \tilde{A}; \theta)$ are the mean and standard deviation of the latent variable Z obtained from the encoder p with parameters θ . The symbol \odot denotes Hadamard product, and ε is a noise term sampled from a standard normal distribution $\mathcal{N}(0, I)$. The parameter α scales the strength of the noise. This formulation captures the distribution of the latent variables conditioned on the input data X, \tilde{A} .

We construct the GIN-based encoder with the same parameter as Arch2Vec [4], which is a 5-layer GIN with hidden dimensions of 128, 128, 128, 128, 16. The L -layer GIN is used to obtain the node feature map H and can be formulated as follows:

$$H^l = MLP^l \left((1 + b^{(l)}) \cdot H^{(l-1)} + \tilde{A}H^{(l-1)} \right), l = 1, 2, \dots, L, \quad (5)$$

where $H^0 = X$ represents the input node features, b is the bias term, and MLP is a multi-layer perceptron. Each MLP layer consists of a linear layer with batch normalization, and in our approach we use ReLU activation function. The node embedding H^L obtained from the last GIN layer is then converted to mean μ and standard deviation σ of the posterior approximation $p(Z|X, \tilde{A}; \theta)$ in Equation (4). Throughout this process, the node representations are transformed into vectors with the same number of channels as the hidden dimension of the last GIN layer. Therefore, the dimension of the mean latent representation is $(|V|, 16)$, where $|V|$ represents the



number of vertices in the graph G and 16 is the output channels of the GIN.

2. Decoder

Graph generation is a complex problem, requiring determination of the number of a node's neighbors and the nodes it connects to, several works employ RNN methods to generate graphs by iteratively generating nodes and edges. In our approach, we utilize a multi-head transformer-based model with 3 transformer blocks. The embedded dimension is 32, and we set the hidden dimension to 256 for the Feed Forward network (FFN) within the transformer blocks, serving as our graph generator. We also incorporate positional encoding to help the model grasp the positional information of each node. The transformer model's self-attention mechanism allows it to consider node features across all node embeddings from the graph encoder we introduced earlier. The transformer-based graph decoder decodes the encoded latent representation into two sections: an operation one-hot matrix and an adjacency matrix. Both sections are treated as classification tasks.

After passing through the transformer blocks, the input will be reshaped to $(|V|, Q_{dim})$, where we set $Q_{dim} = 32$. Subsequently, it goes through a fully connected layer followed by a softmax function that project it to a shape of $(|V|, |OPS|)$ for the operation part. For the adjacency matrix section, it will feed into two layers: one to project it to $(|V|, |V|)$ fit the adjacency matrix's shape, and the second is a linear layer followed by a softmax activation to $(|V|, |V|, 2)$ for classification. The operation part selects the



operation type for each node, while the adjacency matrix part predicts the existence of an edge.

Therefore, the tensor shape of operations is $(|V|, |OPS|)$, where $|OPS|$ represents the size of the candidate operation set. The tensor shape of the adjacency matrix is $(|V|, |V|, 2)$, which is view as a binary classification task. Given the latent representation Z obtained from the encoder with a shape of $Z \in \mathbb{R}^{|V| \times 16}$, the decoder Q can be formulated as follows:

$$Q(Z) = [Nodes_{OPS}, Adj_{Matrix}], \quad (6)$$

$$Nodes_{OPS} = FC_{ops}(Transformers(Z)), \quad (7)$$

$$Adj_{Matrix} = FC_{adj1}(FC_{adj0}(Transformers(Z))), \quad (8)$$

here FC denotes the fully connected layers and $Transformers$ refer to the transformer blocks. Finally, we can obtain the model architecture from the reconstructed graphs.

3. Invertible Neural Network

An overview of the Invertible Neural Network (INN) can be seen in Figure 4. During the training process, the latent representation x is obtained from the graph encoder and is flattened to have a dimension of $|V| \cdot 16$. The input and output dimensions of the INN must be the same. In our case, the output y has a dimension of 1, representing the architecture accuracy. The remaining dimensions of the output are $|V| \cdot 16 - 1$ for z , which is obtained from a normal distribution and is independent of y .



We utilize the implementation of the INN introduced by [10] and adjusts certain hyperparameters. In our work, we employ an INN with two-coupling layers and 4 hidden layers, each with a hidden dimension of 128.

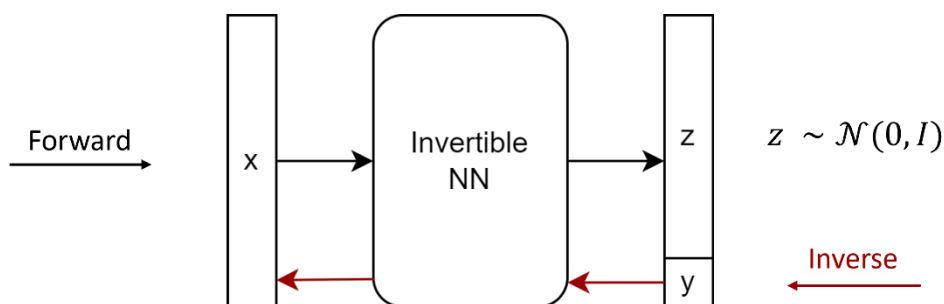
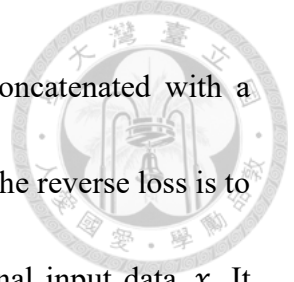


Figure 4: The forward and inverse processes of INN.

During training, the Invertible Neural Network (INN) is trained on both the forward and inverse tasks. In the forward training process, the input data is denoted as x . The objective is to minimize the regression loss, which quantifies the difference between the predicted output y and the corresponding true label. Additionally, the distance between $z|y$ and a normal distribution is minimized using a function that quantifies the dissimilarity between two probability distributions.

The reason for concatenating z with y when calculating the gradient of z is to ensure that z and y are independent for convergence. This condition guarantees that $p(z|y)$ is equal to the prior distribution $p(z)$. Consequently, during the inverse process, y can be successfully inverted by sampling different z values from the normal distribution.



During the training of the inverse process, the y value is concatenated with a randomly sampled normal distribution z as input. The objective of the reverse loss is to quantify the dissimilarity between the output data y and the original input data x . It serves as a measure of how well the INN model can reverse from output data to the input data. By minimizing the reverse loss, the model aims to enhance its ability to accurately reconstruct the original input from the obtained output.

C. Training Method

1. Objective function

(a) Graph Variational Autoencoder

The Graph Variational Autoencoder (GVAE) aims to reconstruct the input graph, with regularization of the latent representation achieved through minimization of the KL-divergence. The loss function for the GVAE is defined as:

$$L_{GVAE} = \alpha L_{nodes} + \beta L_{edges} + \gamma L_{KL}. \quad (9)$$

The terms L_{nodes} and L_{edges} terms, which measure the reconstruction loss for the node features and adjacency matrix, respectively. In our implementation, we utilize cross-entropy loss for both L_{nodes} and L_{edges} . The tradeoff weights α and β between these terms are determined by hyperparameters, which in our case are set to 1.0 for both L_{nodes} and L_{edges} .

The L_{KL} term represents the KL-divergence between the learned latent distribution



and the prior distribution, which serves as a regularization term. In our implementation, the weight for this term, γ , is set to 0.16.

By optimizing the L_{GVAE} objective function, the GVAE model is able to reconstruct the input graph while ensuring a meaningful and regularized latent representation. The architecture of the GVAE model is visualized in Figure 5.

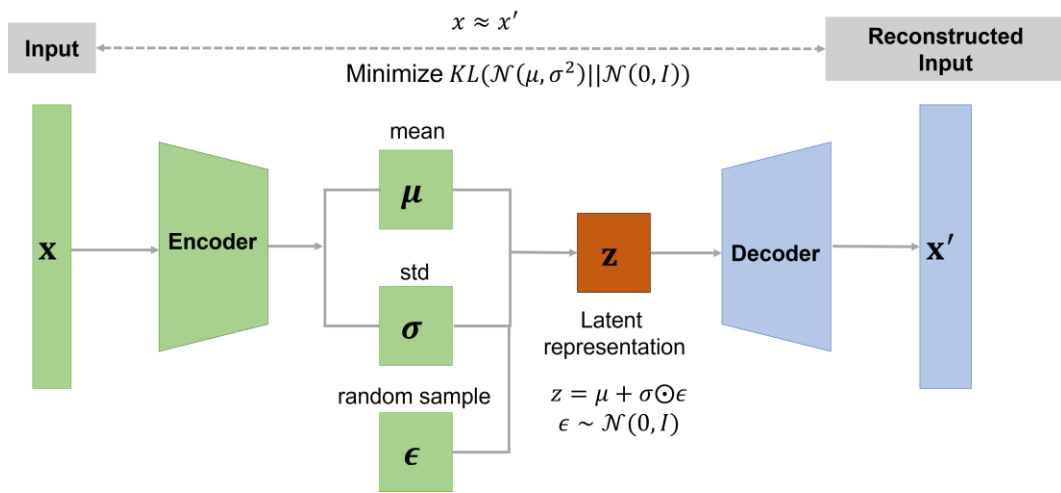
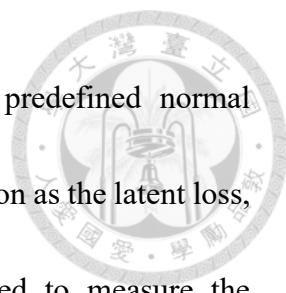


Figure 5: GVAE

(b) Invertible Neural Network

In our implementation, the loss function for an invertible network consists of three terms: $L_{INN} = L_{reg} + L_{rev} + L_{latent}$. The first term, L_{reg} , is supervised regression term. The mean squared error (MSE) loss is used in our implementation to measure the difference between the predicted accuracy and the true accuracy of the architectures. The second term, L_{rev} , is an unsupervised reverse loss. It measures the difference between the inverse result and the input data. The final term, L_{latent} , is an unsupervised latent




loss. It ensures that the predicted latent variable z follows a predefined normal distribution. We use the Maximum Mean Discrepancy (MMD) function as the latent loss, but other loss functions such as KL-divergence can also be used to measure the distribution distance.

To balance the contribution of these terms, we set tradeoff weights. Specifically, we assign weights of 5.0, 10.0, and 1.0 to L_{reg} , L_{rev} , and L_{latent} , respectively. These weights determine the relative importance of each term in the overall loss function.

2. Pre-train GVAE

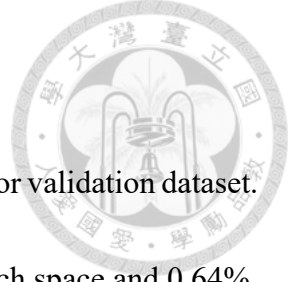
In our approach, we initially pretrain our Graph Variational Autoencoder (GVAE) model with unlabeled architecture graph data from the search space. This pretraining step enables the decoder to reconstruct the architecture graphs from the latent representations, leveraging a large amount of unlabeled data available in the search space. This self-supervised training mechanism is efficient since it does not require training actual architecture which would require a time-consuming process. In scenarios where there is no tabular benchmark available for querying architecture performance, training the actual architectures would require significant time and resources. By pretraining the GVAE model, the encoder can extract meaningful features from the unlabeled architectures, which can be beneficial for downstream tasks. We partition the architecture datasets into three subsets. 80% of the total data is allocated for training, 10% for validation, and the



remaining 10% for testing. Subsequently, we train our GVAE model for 500 epochs. We set the batch size to 64 and use the Adam optimizer. The learning rate is set to $1e-3$, adhering to the default setting for Tensorflow machine learning framework. We employ the ReduceLROnPlateau learning rate scheduler which is provided by Tensorflow and we set the factor to 0.1. This means the learning rate multiplied by 0.1 when reduced. We set the patience parameter to 50; thus, if there is no improvement in the validation loss for 50 consecutive epochs, the learning rate is subsequently reduced. We employ an early stopping technique for 100 patience that monitors the validation loss. The training process will be terminated if the validation loss does not show any improvement for 100 consecutive epochs. Additionally, we restore the weights corresponding to the best validation loss achieved during training.

3. Fine-tune INN

Next, we have two options for the fine-tuning method. The first option is partial fine-tuning, where we freeze the weights of the GVAE, and perform partial fine-tuning of the INN's weights via gradient descent. The second option is end-to-end fine-tuning, where both weights of GVAE and INN are fine-tuned using the gradient descent method. Although the partial fine-tuning process is more efficient than the end-to-end fine-tuning process, as it involves fewer number of weights requiring training, the end-to-end fine-tuning approach performs well empirically. We will compare these two fine-tuning



methods in our experiments.

For the initial training, we set it to 50 for training dataset and 50 for validation dataset.

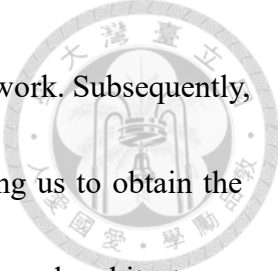
This dataset size corresponds to 0.023% of the NAS-Bench-101 search space and 0.64% of the NAS-Bench-201 search space. In the INN training pipeline, it is necessary to predict the random variable z to match a predefined distribution, which is normal distribution in our case. However, a training set of 50 data might too small for INN converging, so we augment the training data during the fine-tuning process by simply repeating the data by a factor. We set this factor to 20 in our approach, so the size of initial training dataset becomes $50 \times 20 = 1000$ data in total.

D. Retrain and Search

1. Algorithm

In our approach, the retraining and searching procedures are performed iteratively. We begin by performing a search for well-performing architectures by inversely mapping the highest accuracy (1.0) to the corresponding architectures. These architectures are then added to the training dataset for retraining. This iterative procedure continues until a predefined query budget or maximum number of retraining runs is reached. The final search results are reported by selecting the top-performing architecture from our records.

The details of this procedure can refer to **Algorithm 1**. In each search iteration, we first generate 200 unseen neural architectures by inversely mapping (1.0) to the latent



representation of corresponding architectures via invertible neural network. Subsequently, the latent representation is decoded using the graph decoder, allowing us to obtain the corresponding neural architectures. Now we have 200 candidates of neural architectures.

How do we select which of them to query the true accuracy from tabular benchmark and add them to the training dataset? We use our graph encoder coupled with the INN to predict the accuracy of these candidates. This step does not use the query budget until we perform the query from the tabular benchmark. We then select top- k predicted accuracy candidates to query the true accuracy of them from the tabular benchmark and include them in the training dataset for retraining. In our approach, we select top-5 candidates in each iteration.

Because the model gradually converges to a local or global optimal region, the diversity of generated architectures from the decoder decreases. As a result, it becomes challenging to find new candidates of neural architectures. To address this issue, if the decoder fails to generate enough number of new architectures, we add some noise from a normal distribution with a mean of 0 and a standard deviation of α to the latent representation of the neural architecture before feeding it into the decoder. The range of α is 0.0 to 0.1, and we increase α incrementally until we can collect 100 candidates in each searching phase or reach the maximum generation budget. This approach helps to maintain exploration and discover of new architecture candidates during the search

process.

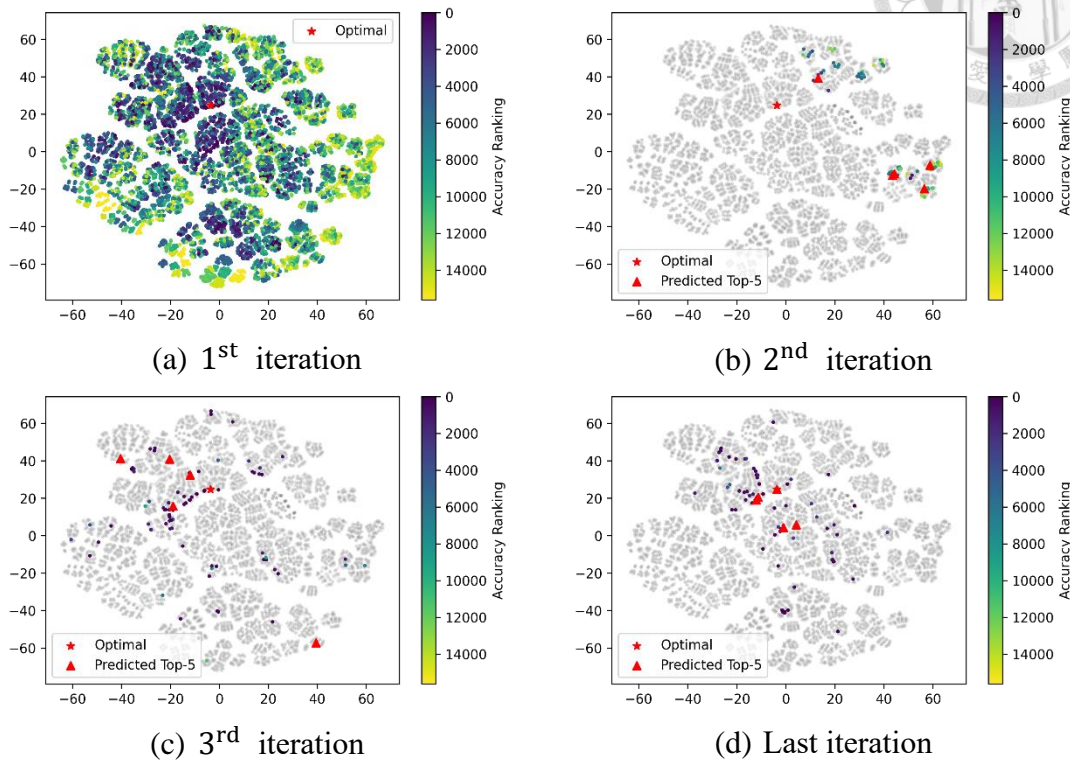


Figure 6: This visualization, created via t-SNE, demonstrates the retraining and searching process for each iteration on CIFAR-10 of the NAS-Bench-201 search space. The 100 candidates, generated by inversely mapping 1.0, are color-coded, while the non-selected neural architectures are displayed in grey.

2. Rank-based Weighted Loss

The rank-based weighted loss is inspired by the generative Latent Space Optimization (LSO) technique used in AG-Net [9]. The main principle of this approach is to pay more attention to higher accuracy data, and pay less attention to lower accuracy data. We achieve this by multiplying a weight to the loss value of each data point based on its rank of accuracy. The formula for the rank based weight is proposed by [24] and we adapt it for our method. Our rank-based weight function can be defined as follows:

$$w_i(x_i, B, k) = \frac{1}{k|B| + \text{rank}(x_i, B)}, x_i \in B, i = 1, \dots, |B|, \quad (10)$$

$$\text{rank}(x_i, B) = |\{x_j: \text{accuracy}(x_j) > \text{accuracy}(x_i), x_j \in B\}|, \quad (11)$$

where x_i is an architecture graph data within a mini-batch B and k is a hyperparameter that controls the smoothness of the weights. We set k to 1e-3 which is similar to [9].

In this approach, the model incrementally trains on higher-accuracy data during the iterative retraining and searching process. While high-accuracy data is more important than low-accuracy data in NAS tasks, our model does not need to be capable of predicting all of the data in the search space. Instead, it focuses on high-accuracy region. We will assess the impact of this technique in the ablation studies section.

Algorithm 1: Search and Retrain

input: (i) Encoder E , Decoder D and Invertible neural network INN

input: (ii) Query budget b and Maximum runs r

input: (iii) Retrain epochs e

input: (iv) Training dataset DT and Validation dataset DV

```
1   $iteration \leftarrow 0$ 
2  while  $|DT \cup DV| < b$  do
3       $G_{cand} \leftarrow \{\}$ ;
4      /* Generate 100 candidates by inverse 1.0 */
5      while  $|G_{cand}| < 100$  do
6           $z \sim N(0, I)$ ;
7           $g \leftarrow D(\text{inverse}(INN, 1.0, z))$ ;
8          if  $g \notin DT \cup G_{cand}$  then
9               $G_{cand} \leftarrow G_{cand} \cup \{g\}$ ;
10         end
11         /* Predict accuracy of architectures in  $G_{cand}$  by using  $INN$ ,  $E$  and
12            then select top- $k$  candidates */
13          $G_{cand} \leftarrow \text{select}(G_{cand}, E, INN, k)$ ;
14         /* Query the true label and add them to training set */
15          $DT \leftarrow DT \cup \text{eval}(G_{cand})$ ;
16          $\text{train}(E, D, INN, DT, DV, e)$ ;
17          $iteration \leftarrow iteration + 1$ ;
18         if  $iteration \geq r$  then
19             break;
20     end
```



Chapter 4 Experiments



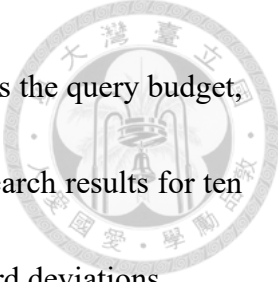
We apply our neural architecture method on two tabular NAS benchmarks, including NAS-Bench-101 [12] and the NAS-Bench-201 [13]. The experimental results show that our method is comparable to other state-of-the-art NAS methods. The experiment is divided into three parts. First, we introduce the metrics in our experiments, which include the ability to conduct the architecture search, regression and inversion. Second, we provide the neural architecture search results, regression results, and a visualization of inversion ability on the NAS-Bench-101 benchmark. We compare the architecture search results with those from other studies. Finally, we also present the results of the aforementioned tasks on the NAS-Bench-201 benchmark. In the regression and inversion experiments, to better illustrate our concept of using the Invertible Neural Network (INN) to obtain neural architectures from their accuracy, we train a model using 350 training data and 50 validation data for the regression and inversion experiments.

A. Evaluation Metrics

This section introduces the evaluations we conducted on both NAS benchmarks.

1. Architecture Search

The evaluation of architecture search is determined by how accurately (i.e., with high accuracy) the neural architectures can be found given a certain query budget or wall time. We use the same settings for both NAS benchmarks, including fine-tuning method



and the use of rank-based weighted loss. The only adjustable setting is the query budget, which we modify for comparisons with other works. We report the search results for ten independent runs in our experiments, including the means and standard deviations.

2. Regression

In the evaluation of regression, we plot the graph of $y_{predict}$ against y_{true} to illustrate the correlation between predicted accuracy $y_{predict}$ and true accuracy y_{true} of the neural architectures. This demonstrates how well our model can handle the regression task with unseen data from the testing dataset.

3. Inversion

We use the re-simulation concept in our inversion evaluations. The re-simulation error measures the difference between the accuracy used as input for the Invertible Neural Network (INN) and the true accuracy of the architecture, which is the inverted output from the INN.

The inversion experiment is divided into two parts. First, we use the accuracy from the datasets as input for inversion. Second, we use a range from 0.0 to 1.0, with a step of 0.005, as inversion input. For visualization this experiment, we use the accuracy y' as the input for our INN model and invert it to the latent representation. Subsequently, we use the decoder to decode the latent representation and obtain the neural architecture x .

We query the true accuracy y of x from the tabular benchmark and plot the y', y



graph to measure the inversion capability of our model.

B. NAS-Bench-101

1. Architecture Search

We report the best accuracy of architectures we found under the query budget of 190, 300, 500 in Table 1. We compare our results to other methods, including Arch2Vec [4], AG-Net [9], CR-LSO [7] and traditional genetic or optimization algorithms which served as baseline methods. These include BANANAS [11], Random Search (RS) [25], Local Search (LS) [26, 27], Regularized Evolution (RE) [28].

Our results outperform the baseline methods and CR-LSO, which is the state-of-the-art method on the ImageNet16-120 dataset of NAS-Bench-201. However, the performance of CR-LSO on NAS-Bench-101 is not as good as its performance on NAS-Bench-201. Additionally, our results also comparable to the results of AG-Net.

Table 1: The comparison results on NAS-Bench-101. The means and standard deviations are reported.

Method	Val. Acc	Val. StD	Test Acc	Test StD	Queries
Optimal*	95.06	-	94.32	-	
BANANAS	94.69	0.08	94.14	0.11	192
RS	94.19	0.50	93.47	0.47	192
LS	94.67	0.21	93.73	0.44	192
RE	94.22	0.27	93.63	0.35	192
AG-Net	94.90	0.22	94.18	0.10	192
Ours	94.62	0.22	94.02	0.14	190
BANANAS	94.76	0.13	94.16	0.11	300
AG-Net	94.96	-	94.20	-	300

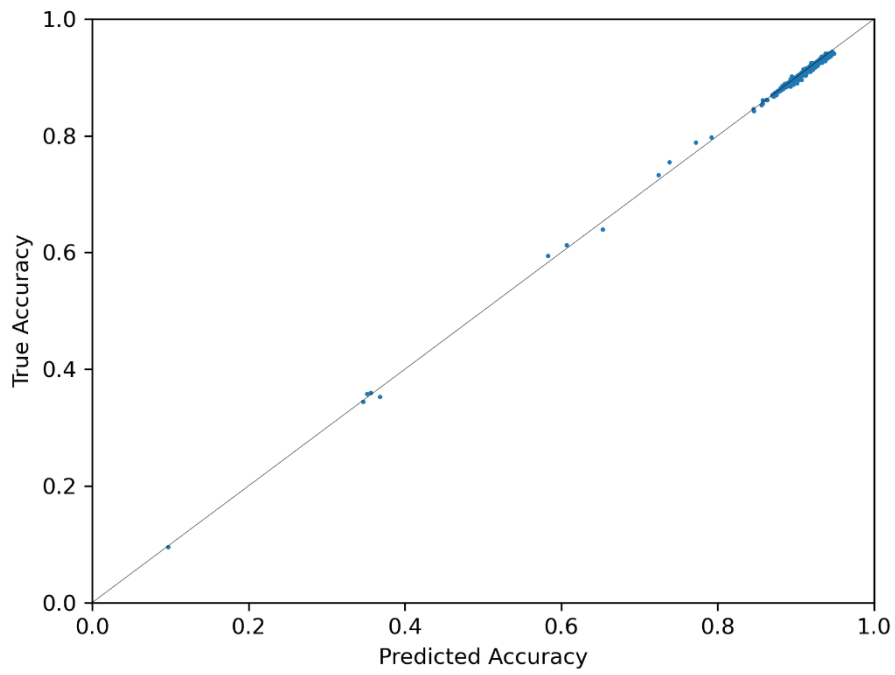
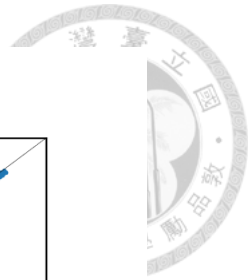
Ours	94.96	0.18	94.21	0.05	300
Arch2Vec-RL	-	-	94.10	-	400
CR-LSO	-	-	93.97	2e-3	500
BANANAS	94.79	0.13	94.16	0.11	500
AG-Net	94.97	0.16	94.20	0.07	500
Ours	95.03	0.10	94.22	0.04	500

2. Regression

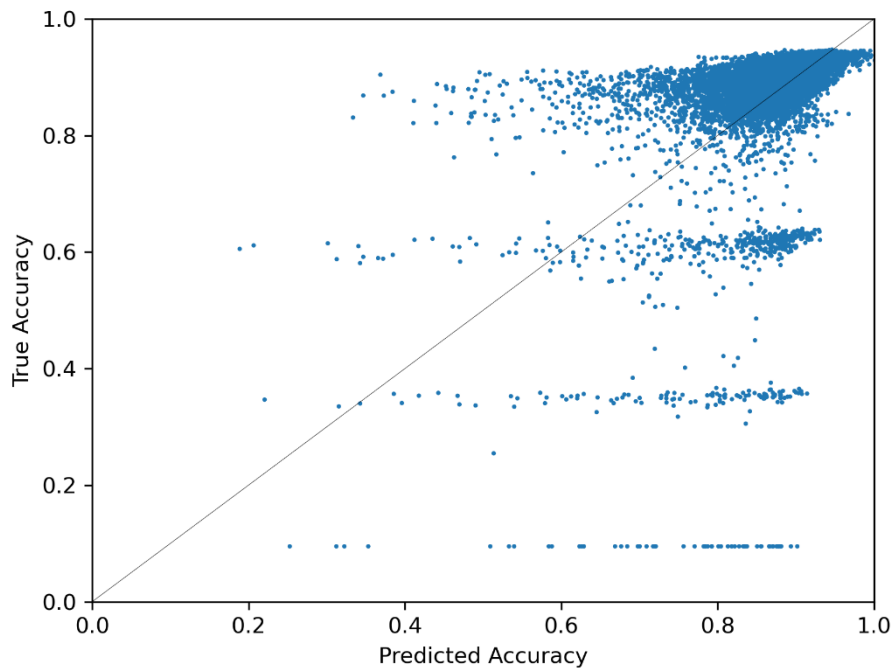
In this experiment, we use 350 data points for the training set, 50 data points for the validation set, and 10% of the entire search space data for the testing set. The visualization results are provided on Figure 7. We can observe our proposed model architecture can fit the regression tasks even on this large search space (423K), by using approximately 0.1% of the data.

3. Inversion

We have two parts of inversion experiment. First is use the accuracy from the datasets as input for inversion and second is use a range from 0.0 to 1.0, with a step of 0.005, as inversion input. The results of first part are provided on Figure 8, and the second part is shown in Figure 9.

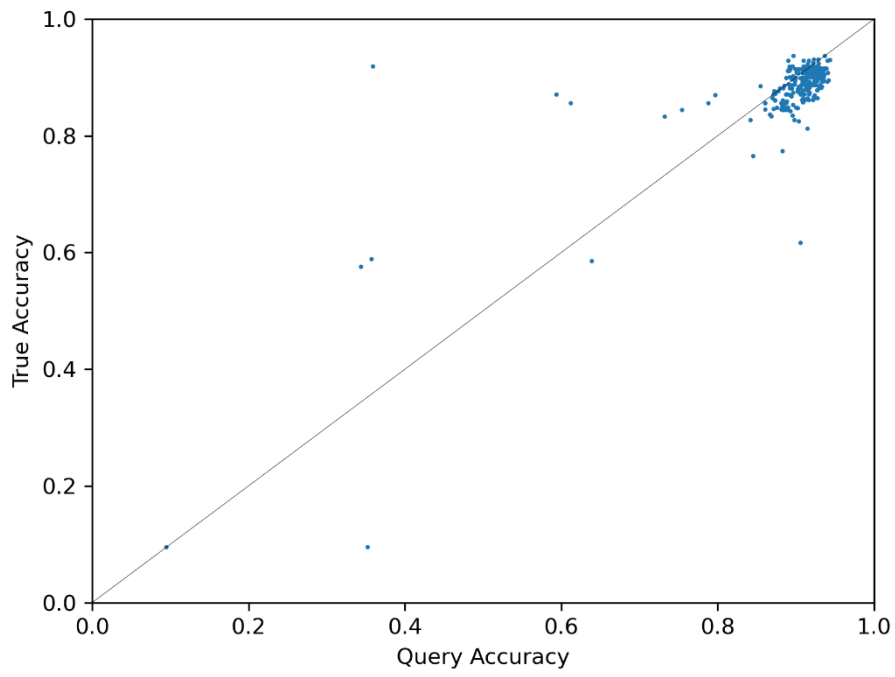


(a) Training set

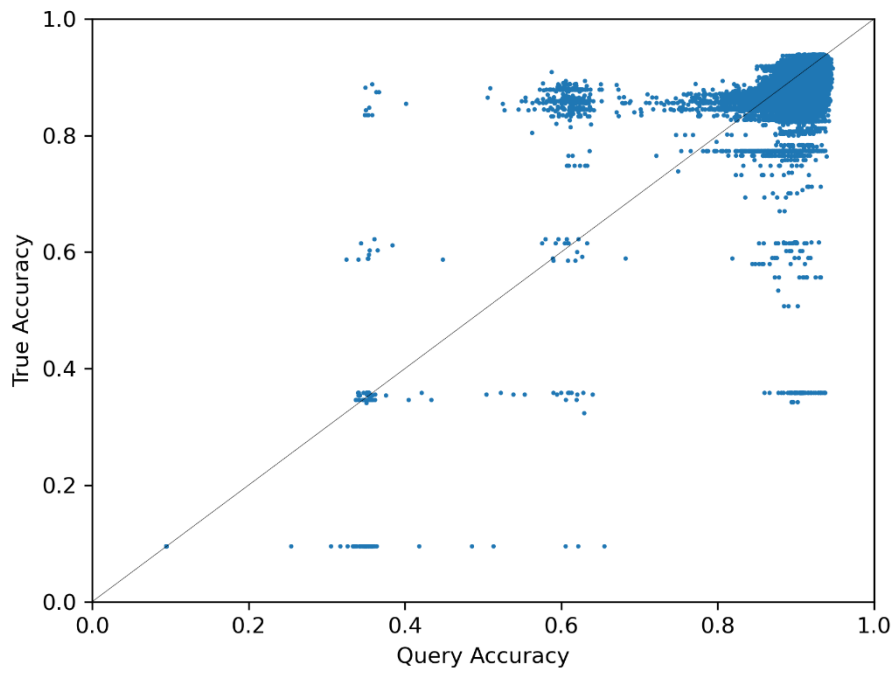


(b) Testing set

Figure 7: Regression results on NAS-Bench-101.



(a) Training set



(b) Testing set

Figure 8: Inversion results on NAS-Bench-101

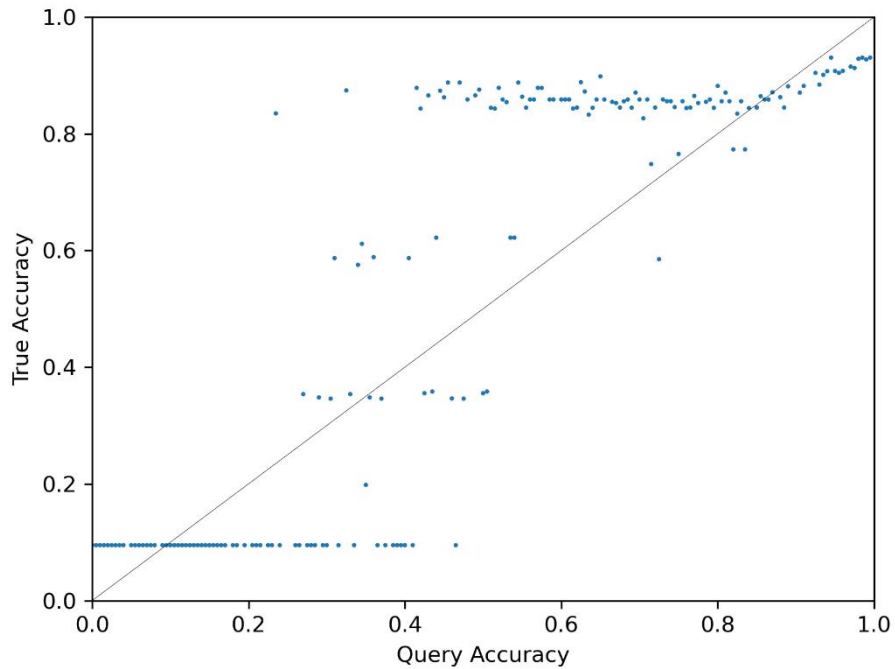


Figure 9: Inversion result over a range on NAS-Bench-101.

C. NAS-Bench-201

1. Architecture Search

We present the accuracy of architectures we discovered under query budget of 190 and 400, on three datasets from NAS-Bench-201 benchmark, including CIFAR-10, CIFAR-100 and ImageNet16-120. We compare our results to other state-of-the-art methods, including AG-Net [9], CR-LSO [7] and traditional genetic or optimization algorithms which served as baseline methods. These include BANANAS [11], Random Search (RS) [25], Local Search (LS) [26, 27], Regularized Evolution (RE) [28].

In Table 2, we demonstrate our neural architecture search results alongside several baseline methods on NAS-Bench-201 search space. With a query budget of 190, our



approach is comparable to the AG-Net on three datasets, and outperforms AG-Net in terms of test accuracy on the ImageNet16-120 dataset. Additionally, the cost of our methods (measured in GPU hours) is only one-tenth of AG-Net. Our result on test accuracy of ImageNet16-120 dataset is superior to CR-LSO, which is the state-of-the-art method on this dataset, even though we use an extremely lower query budget (190 vs 500). Furthermore, we can identify the global optimal neural architectures on CIFAR-10 and CIFAR-100 datasets. For more details, we provide the means and standard deviations of ten independent trials in Table 7 of Appendix A.

Table 2: The comparison results on NAS-Bench-201. The means are reported.

Method	CIFAR-10		CIFAR-100		ImageNet16-120		Queries	GPU hours
	Val.	Test	Val.	Test	Val.	Test		
Optimal*	91.61	94.37	74.39	73.51	46.73	47.31		
BANANAS	91.55	94.26	73.49*	73.51*	46.68	46.49	192	0.05
RS	91.27	94.02	72.12	72.31	45.67	46.08	192	0.01
LS	91.53	94.31	73.28	73.25	46.44	46.77	192	0.01
RE	91.48	94.94	72.86	72.98	46.04	46.43	192	0.01
AG-Net	91.60	94.37*	73.49*	73.51*	46.64	46.43	192	5
Ours	91.60	94.37	73.49*	73.51*	46.61	46.98	190	0.5
CR-LSO	91.54	94.35	73.44	73.47	46.51	46.98	500	0.13
AG-Net	91.61*	94.37*	73.49*	73.51*	46.73*	46.42	400	5.2
Ours	91.61*	94.37*	73.49*	73.51*	46.70	47.20	400	1.2

2. Regression

We provide the visualization results of CIFAR-10 on Figure 11. The results for the remaining two datasets are provided on Figure 15 of Appendix A. We can see our model



fits the regression tasks on these datasets well. In this experiment, we use 350 data points for the training set, 50 data points for the validation set, and 10% of the entire search space data for the testing set.

3. Inversion

We have two parts of inversion experiment. First is use the accuracy from the datasets as input for inversion and second is use a range from 0.0 to 1.0, with a step of 0.005, as inversion input. The results of first part are provided on Figure 12, and the second part is shown in Figure 10. For the results of remaining two datasets can refer to Figure 16 and Figure 17 on Appendix A.

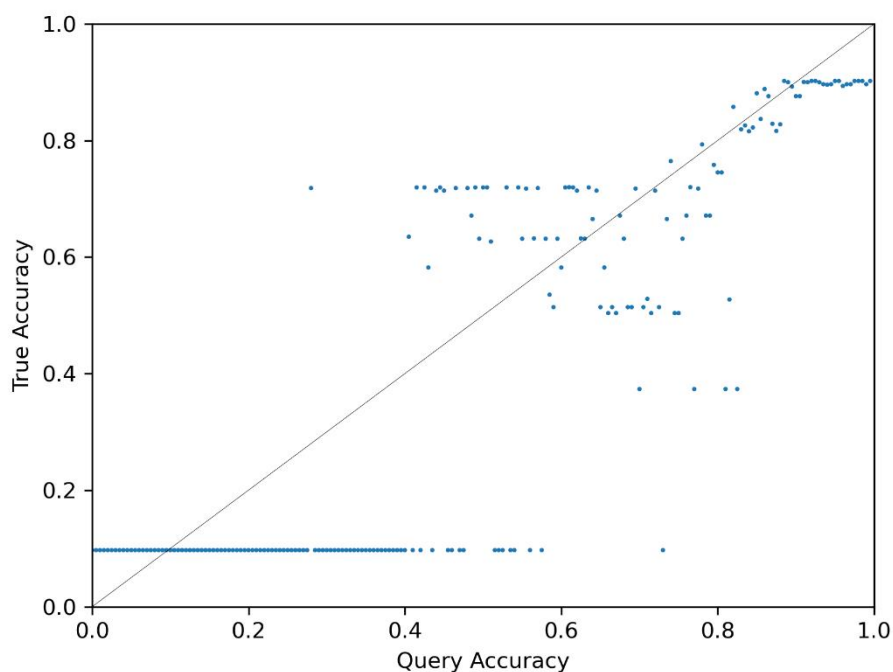
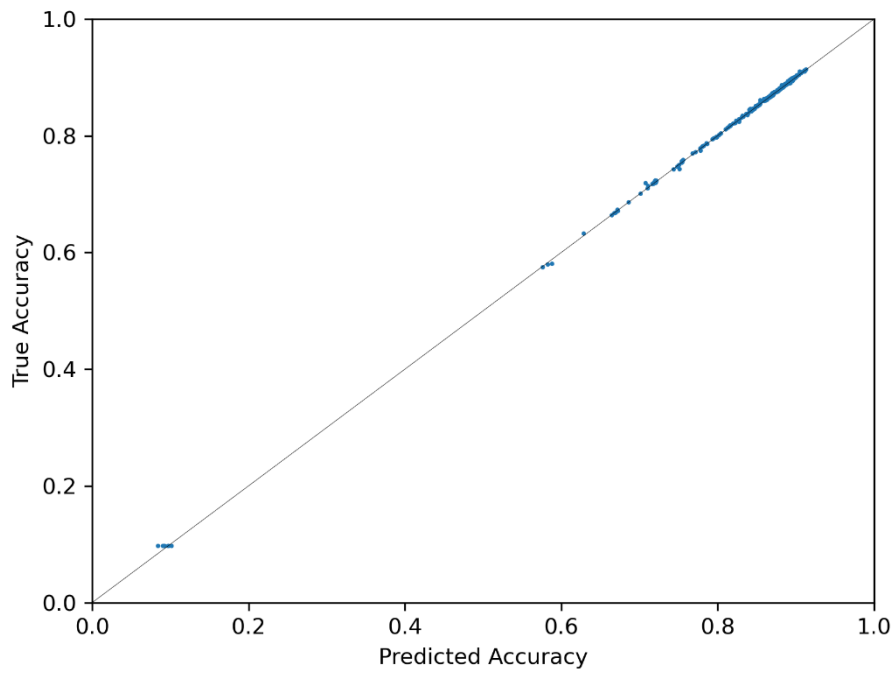
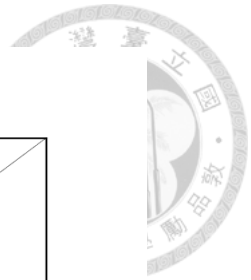
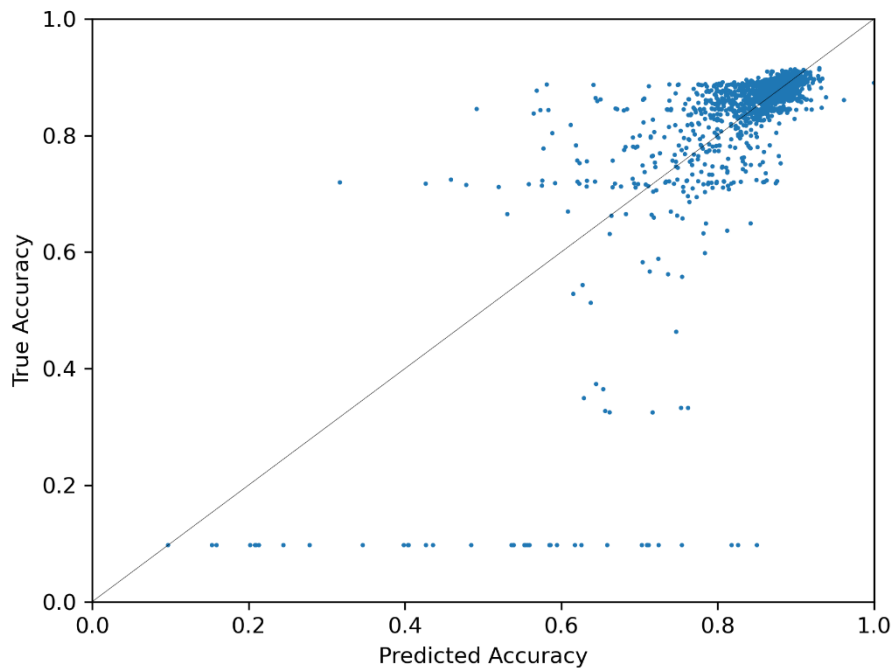


Figure 10: Inversion result over a range on CIFAR-10.



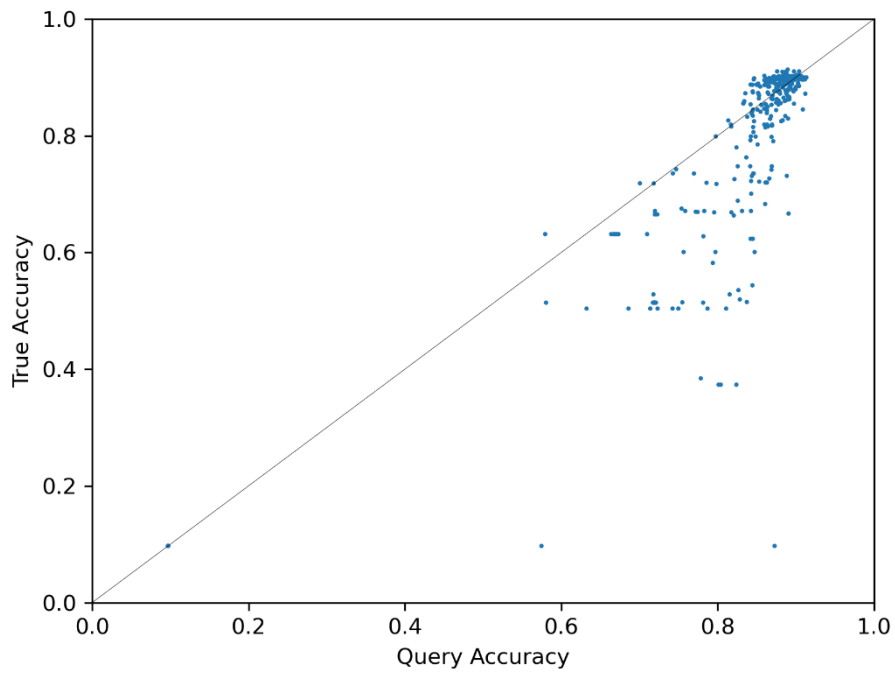


(a) Training set on CIFAR-10

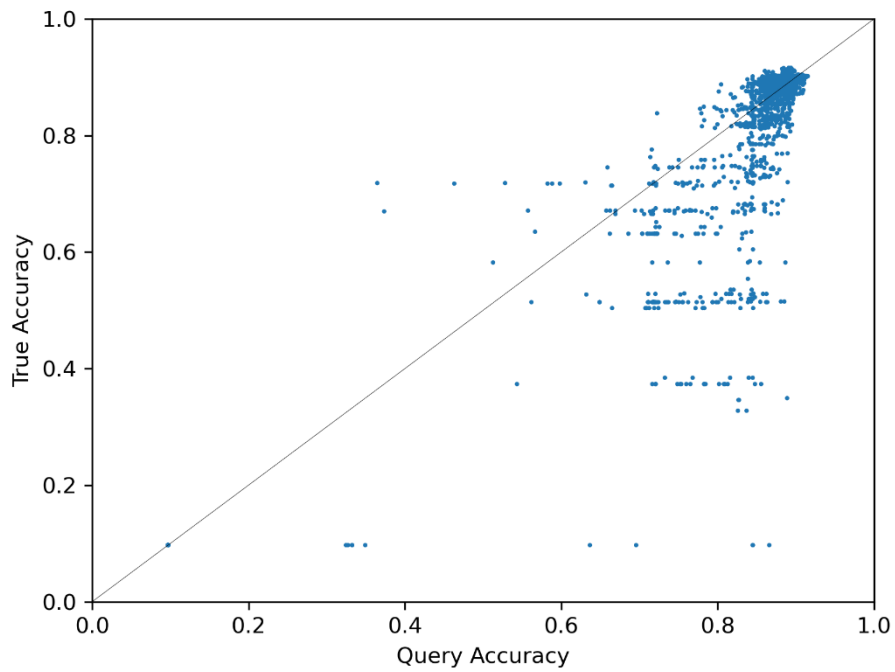


(b) Testing set on CIFAR-10

Figure 11: Regression results on CIFAR-10.



(a) Training set on CIFAR-10



(b) Testing set on CIFAR-10

Figure 12: Inversion results on CIFAR-10.



Chapter 5 Ablation Studies



A. Choice of Candidates Generative Methods

To demonstrate the effectiveness of our concept, which involves using an invertible neural network to obtain high-performing neural architecture candidates by inversely mapping 1.0, and thus enhancing the performance of NAS, we compare our approach with an alternative method that replaces our candidate generation process with random selection. In Table 3 and Table 4, the random selection method, used as a candidate generative approach, performs significantly worse than our method.

Table 3: The comparison of the candidate generative methods on NAS-Bench-101.

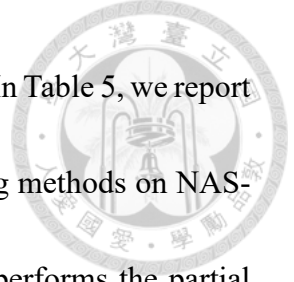
Method	Val. Acc	Val. StD	Test Acc	Test StD	Queries
Optimal*	95.06	-	94.32	-	
Ours-Random	94.41	0.13	93.80	0.11	190
Ours	94.62	0.22	94.02	0.14	190
Ours-Random	94.54	0.09	93.96	0.09	300
Ours	94.96	0.18	94.21	0.05	300

Table 4: The comparison of the candidate generative methods on NAS-Bench-201.

Method	CIFAR-10		CIFAR-100		ImageNet16-120		Queries
	Val.	Test	Val.	Test	Val.	Test	
Optimal*	91.61	94.37	74.39	73.51	46.73	47.31	
Ours-Random	91.36	94.20	72.73	72.97	46.34	46.58	190
Ours	91.60	94.37	73.49*	73.51*	46.61	46.98	190
Ours-Random	91.52	94.33	73.09	73.40	46.55	47.00	400
Ours	91.61*	94.37*	73.49*	73.51*	46.70	47.20	400

B. Choice of Fine-tune Methods

In this section, we will compare the impacts of using the end-to-end fine-tuning and

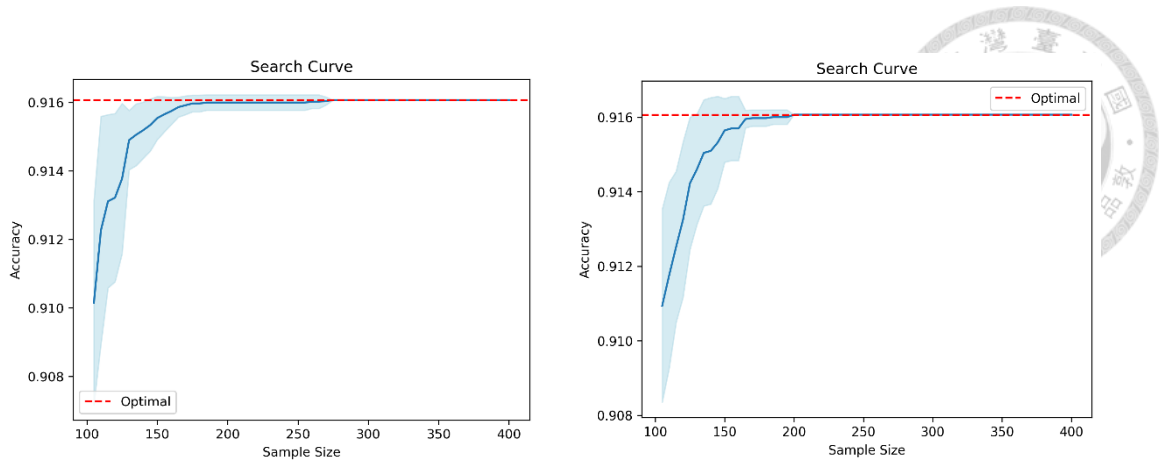


partial fine-tuning methods on the neural architecture search process. In Table 5, we report the mean results of the neural architecture search for both fine-tuning methods on NAS-Bench-201. We can observe the end-to-end fine-tuning method outperforms the partial fine-tuning method in most cases, regardless of the use of rank-based weighted loss.

Table 5: This table compares the two fine-tuning methods. The methods denoted with the postfix 'R' utilize a rank-based weighted loss.

Method	CIFAR-10		CIFAR-100		ImageNet16-120		Queries
	Val.	Test	Val.	Test	Val.	Test	
Optimal*	91.61	94.37	74.39	73.51	46.73	47.31	
Ours-Partial	91.57	94.36	73.40	73.44	46.60	46.83	200
Ours-E2E	91.58	94.34	73.46	73.48	46.54	46.78	200
Ours-Partial-R	91.60	94.37	73.49*	73.51*	46.64	46.98	200
Ours-E2E-R	91.61*	94.37*	73.49*	73.51*	46.62	47.01	200

In Figure 13, we compare the search curves of these two fine-tuning methods, including the use of rank-based weighted loss, using the CIFAR-10 dataset of NAS-Bench-201. We can observe that when employing the end-to-end fine-tuning method, the optimal architecture was discovered with fewer queries.



(a) Partial fine-tuning

(b) End-to-End fine-tuning

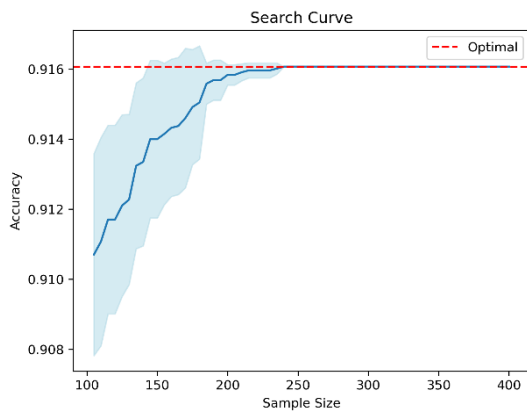
Figure 13: The comparison of search curves for fine-tuning methods with rand-based weighted loss on CIFAR-10 dataset of NAS-Bench-201.

C. Whether Using Rank-based Weighted Loss

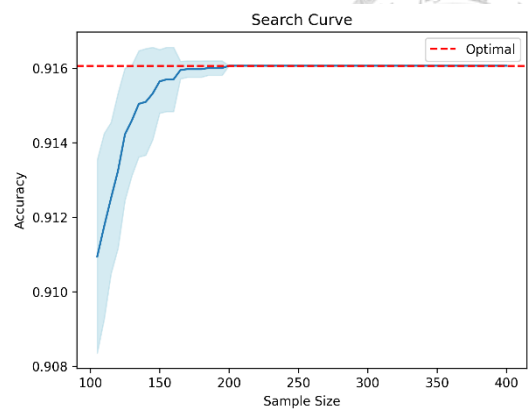
In Table 6, we compare the utilization of rank-based weighted loss. The model, updated with a rank-based weighted loss, can converge to the region of well-performing architectures. According to the results, the application of this technique outperforms all results achieved without using it. Additionally, Figure 14 provides a comparison of the search curves on NAS-Bench-201.

Table 6: A comparison of the use of rank-based weighted loss. Methods with a ‘R’ postfix denote the utilization of rank-based weighted loss.

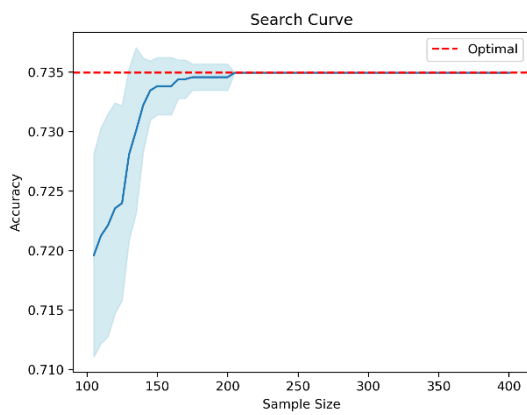
Method	CIFAR-10		CIFAR-100		ImageNet16-120		Queries
	Val.	Test	Val.	Test	Val.	Test	
Optimal*	91.61	94.37	74.39	73.51	46.73	47.31	
Ours-Partial	91.57	94.36	73.40	73.44	46.60	46.83	200
Ours-Partial-R	91.60	94.37	73.49*	73.51*	46.64	46.98	200
Ours-E2E	91.58	94.34	73.46	73.48	46.54	46.78	200
Ours-E2E-R	91.61*	94.37*	73.49*	73.51*	46.62	47.01	200



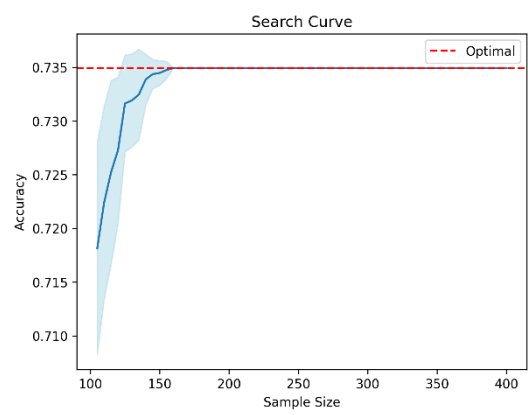
(a) CIFAR-10 w/o weighted loss



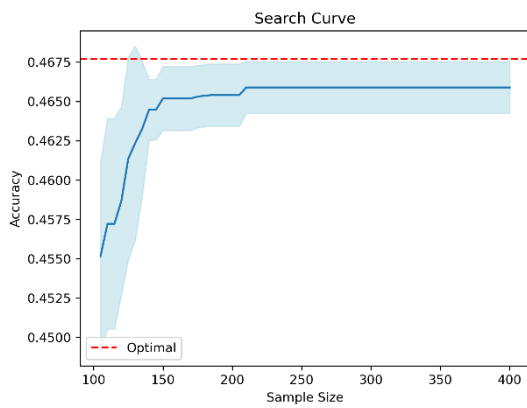
(b) CIFAR-10 with weighted loss



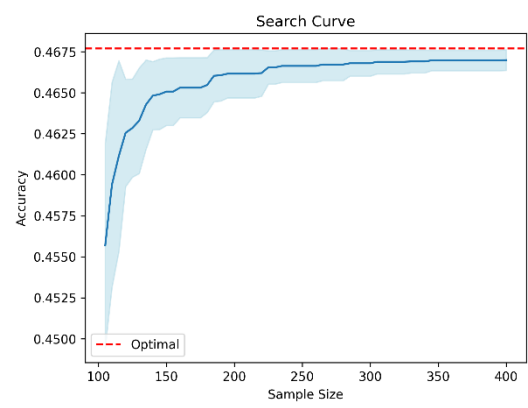
(c) CIFAR-100 w/o weighted loss



(d) CIFAR-100 with weighted loss



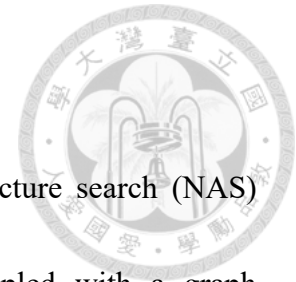
(e) ImageNet16-120 w/o weighted loss



(f) ImageNet16-120 with weighted loss

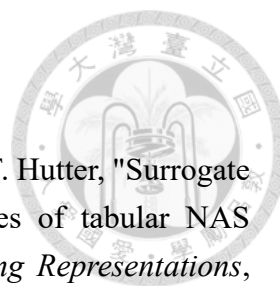
Figure 14: Comparison of search curves with and without the use of rank-based weighted loss on NAS-Bench-201.

Chapter 6 Conclusion



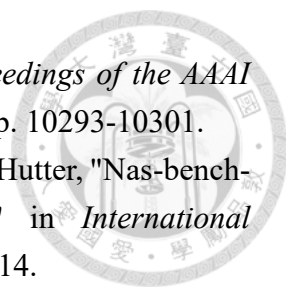
This paper proposes a novel concept to solve neural architecture search (NAS) problems. We employ an invertible neural network (INN) coupled with a graph variational autoencoder to identify the best-performing neural architecture in the latent space by inversely mapping the upper-bound accuracy (1.0) to corresponding neural architectures.

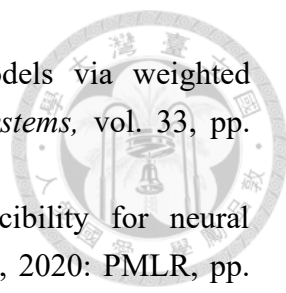
Drawing inspiration from state-of-the-art works, we implement rank-based weighted loss to guide our model to focus on the region of high-performing neural architectures. We then retraining our model iteratively, using an increasing number of high-performing architectures. The experimental results show that our approach outperforms not only the baseline but also some state-of-the-art results on NAS-Bench-101 and NAS-Bench-201 search spaces.



References

- [1] A. Zela, J. N. Siems, L. Zimmer, J. Lukasik, M. Keuper, and F. Hutter, "Surrogate NAS benchmarks: Going beyond the limited search spaces of tabular NAS benchmarks," in *Tenth International Conference on Learning Representations*, 2022: OpenReview. net, pp. 1-36.
- [2] W. Wen, H. Liu, Y. Chen, H. Li, G. Bender, and P.-J. Kindermans, "Neural predictor for neural architecture search," in *Computer Vision—ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXIX*, 2020: Springer, pp. 660-676.
- [3] J. Wu *et al.*, "Stronger nas with weaker predictors," *Advances in Neural Information Processing Systems*, vol. 34, pp. 28904-28918, 2021.
- [4] S. Yan, Y. Zheng, W. Ao, X. Zeng, and M. Zhang, "Does unsupervised architecture representation learning help neural architecture search?," *Advances in Neural Information Processing Systems*, vol. 33, pp. 12486-12498, 2020.
- [5] Y. Tang *et al.*, "A Semi-Supervised Assessor of Neural Architectures," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Los Alamitos, CA, USA, June 2020: IEEE Computer Society, pp. 1807-1816. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/CVPR42600.2020.00188>. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/CVPR42600.2020.00188>
- [6] J. Lukasik, D. Friede, A. Zela, F. Hutter, and M. Keuper, "Smooth Variational Graph Embeddings for Efficient Neural Architecture Search," in *International Joint Conference on Neural Networks, {IJCNN} 2021, Shenzhen, China, July 18-22, 2021*, 2021.
- [7] X. Rao, B. Zhao, X. Yi, and D. Liu, "CR-LSO: Convex Neural Architecture Optimization in the Latent Space of Graph Variational Autoencoder with Input Convex Neural Networks," *arXiv preprint arXiv:2211.05950*, 2022.
- [8] S. S. C. Rezaei *et al.*, "Generative adversarial neural architecture search," *arXiv preprint arXiv:2105.09356*, 2021.
- [9] J. Lukasik, S. Jung, and M. Keuper, "Learning Where To Look—Generative NAS is Surprisingly Efficient," in *Computer Vision—ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXIII*, 2022: Springer, pp. 257-273.
- [10] L. Ardizzone *et al.*, "Analyzing Inverse Problems with Invertible Neural Networks," p. arXiv:1808.04730doi: 10.48550/arXiv.1808.04730.
- [11] C. White, W. Neiswanger, and Y. Savani, "Bananas: Bayesian optimization with

- 
- neural architectures for neural architecture search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021, vol. 35, no. 12, pp. 10293-10301.
- [12] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," in *International Conference on Machine Learning*, 2019: PMLR, pp. 7105-7114.
- [13] X. Dong and Y. Yang, "NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search," in *International Conference on Learning Representations (ICLR)*, 2020. [Online]. Available: <https://openreview.net/forum?id=HJxyZkBKDr>. [Online]. Available: <https://openreview.net/forum?id=HJxyZkBKDr>
- [14] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.
- [15] C. Liu *et al.*, "Progressive neural architecture search," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 19-34.
- [16] B. Deng, J. Yan, and D. Lin, "Peephole: Predicting network performance before training," *arXiv preprint arXiv:1712.03351*, 2017.
- [17] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How Powerful are Graph Neural Networks?," p. arXiv:1810.00826doi: 10.48550/arXiv.1810.00826.
- [18] S. Yan, K. Song, F. Liu, and M. Zhang, "CATE: Computation-aware Neural Architecture Encoding with Transformers," p. arXiv:2102.07108doi: 10.48550/arXiv.2102.07108.
- [19] K. Jing, J. Xu, and P. Li, "Graph Masked Autoencoder Enhanced Predictor for Neural Architecture Search," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, {IJCAI-22}*, L. D. Raedt, Ed., July 2022: International Joint Conferences on Artificial Intelligence Organization, pp. 3114-3120. [Online]. Available: <https://doi.org/10.24963/ijcai.2022/432>. [Online]. Available: <https://doi.org/10.24963/ijcai.2022/432>
- [20] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.
- [21] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61-80, 2008.
- [22] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
- [23] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.
- [24] A. Tripp, E. Daxberger, and J. M. Hernández-Lobato, "Sample-efficient

- 
- optimization in the latent space of deep generative models via weighted retraining," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11259-11272, 2020.
- [25] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *Uncertainty in artificial intelligence*, 2020: PMLR, pp. 367-377.
- [26] T. Den Ottelander, A. Dushatskiy, M. Virgolin, and P. A. Bosman, "Local search is a remarkably strong baseline for neural architecture search," in *Evolutionary Multi-Criterion Optimization: 11th International Conference, EMO 2021, Shenzhen, China, March 28–31, 2021, Proceedings 11*, 2021: Springer, pp. 465-479.
- [27] C. White, S. Nolen, and Y. Savani, "Exploring the loss landscape in neural architecture search," in *Uncertainty in Artificial Intelligence*, 2021: PMLR, pp. 654-664.
- [28] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the aaai conference on artificial intelligence*, 2019, vol. 33, no. 01, pp. 4780-4789.

Appendices



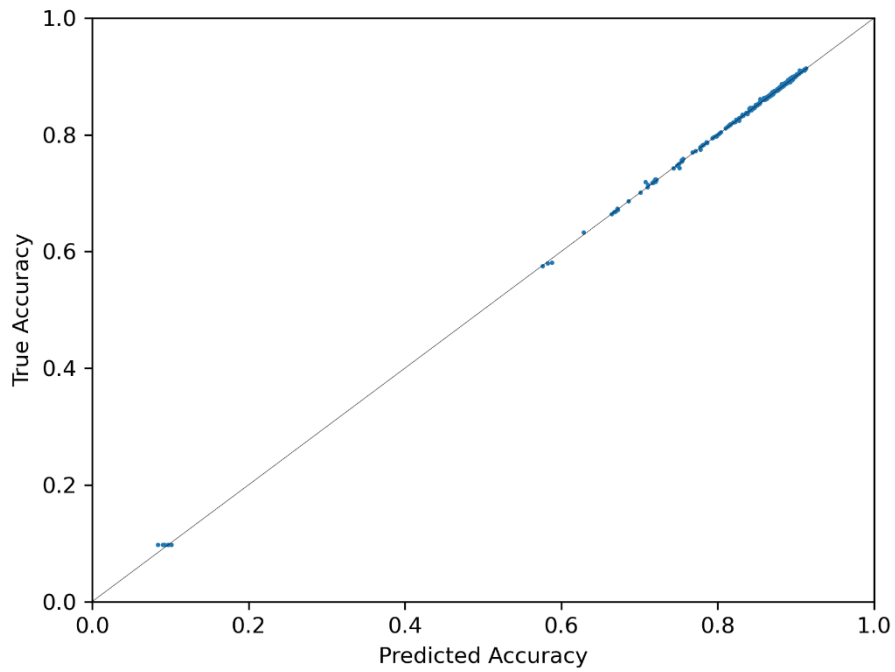
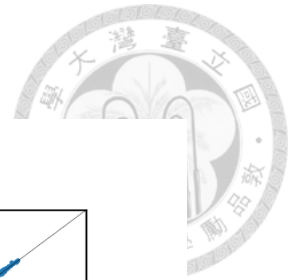
A. NAS-Bench-201

1. Architecture Search

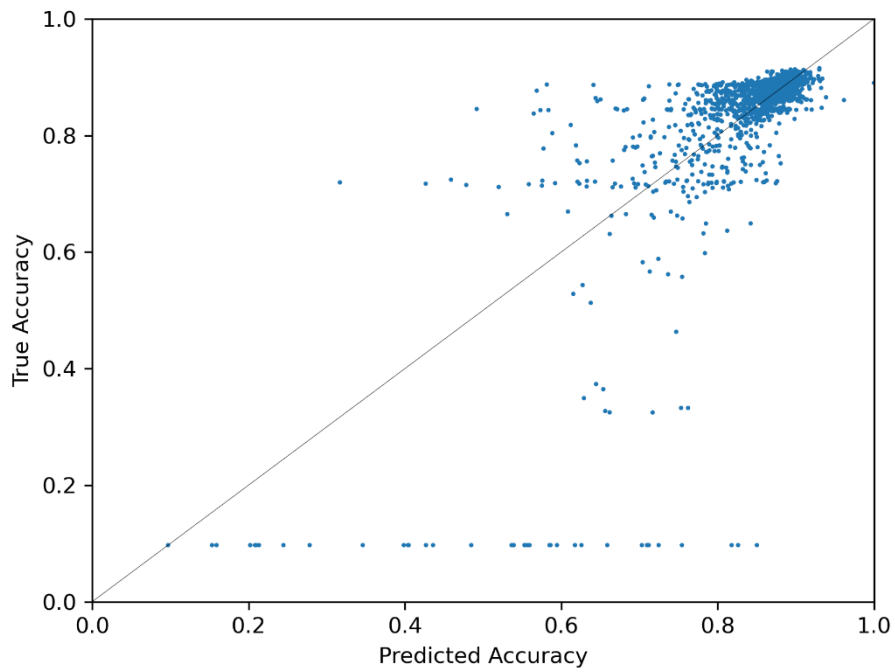
Table 7: The comparison results on NAS-Bench-201. The mean and standard deviation are reported.

Method	CIFAR-10		CIFAR-100		ImageNet16-120		Queries	
	Val.	Test	Val.	Test	Val.	Test		
Optimal*	91.61	94.37	74.39	73.51	46.73	47.31		
BANANAS	Mean	91.55	94.26	73.49*	73.51*	46.68	46.49	192
	StD	0.15	0.22	0.00	0.00	0.09	0.42	
RS	Mean	91.27	94.02	72.12	72.31	45.67	46.08	192
	StD	0.23	0.21	0.90	0.92	0.52	0.60	
LS	Mean	91.53	94.31	73.28	73.25	46.44	46.77	192
	StD	0.15	0.15	0.52	0.58	0.18	0.25	
RE	Mean	91.48	94.94	72.86	72.98	46.04	46.43	192
	StD	0.13	0.21	0.83	0.79	0.54	0.38	
AG-Net	Mean	91.60	94.37*	73.49*	73.51*	46.64	46.43	192
	StD	0.02	0.00	0.00	0.00	0.12	0.34	
Ours	Mean	91.60	94.37	73.49*	73.51*	46.61	46.98	190
	Std	0.02	0.01	0.00	0.00	0.16	0.34	
CR-LSO	Mean	91.54	94.35	73.44	73.47	46.51	46.98	500
	StD	0.05	0.05	0.17	0.14	0.05	0.35	
AG-Net	Mean	91.61*	94.37*	73.49*	73.51*	46.73*	46.42	400
	StD	0.00	0.00	0.00	0.00	0.00	0.00	
Ours	Mean	91.61*	94.37*	73.49*	73.51*	46.70	47.20	400
	Std	0.00	0.00	0.00	0.00	0.06	0.20	

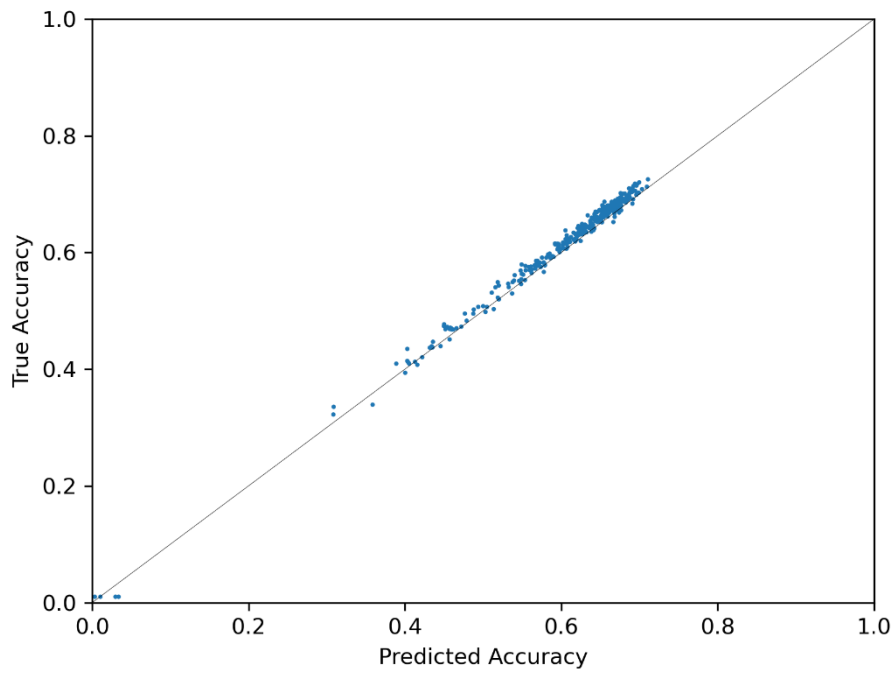
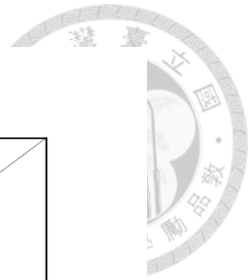
2. Regression



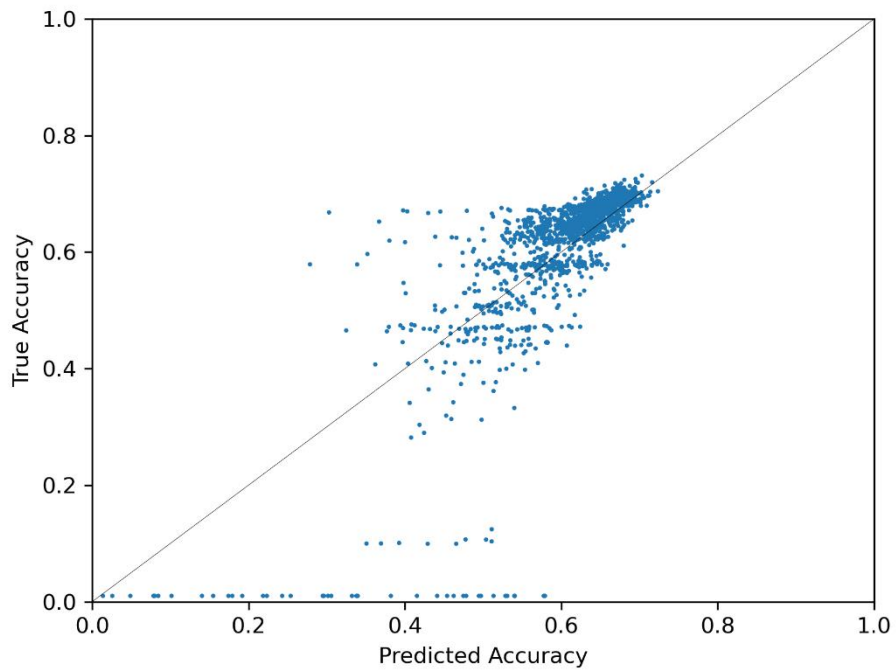
(a) Training set on CIFAR-10



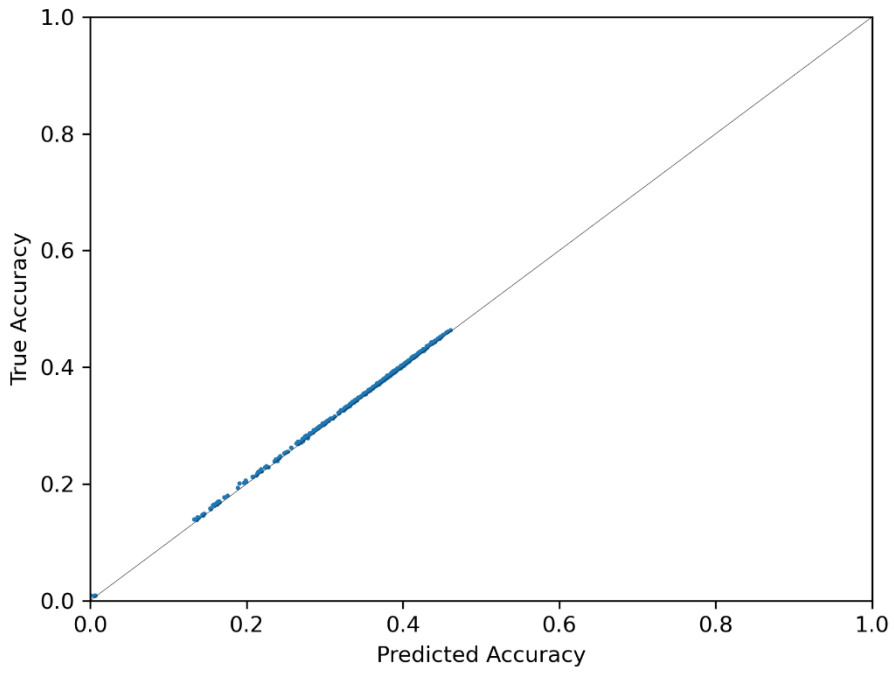
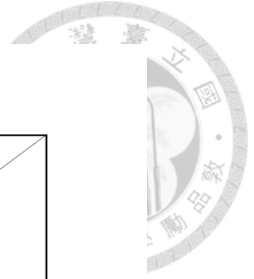
(b) Testing set on CIFAR-10



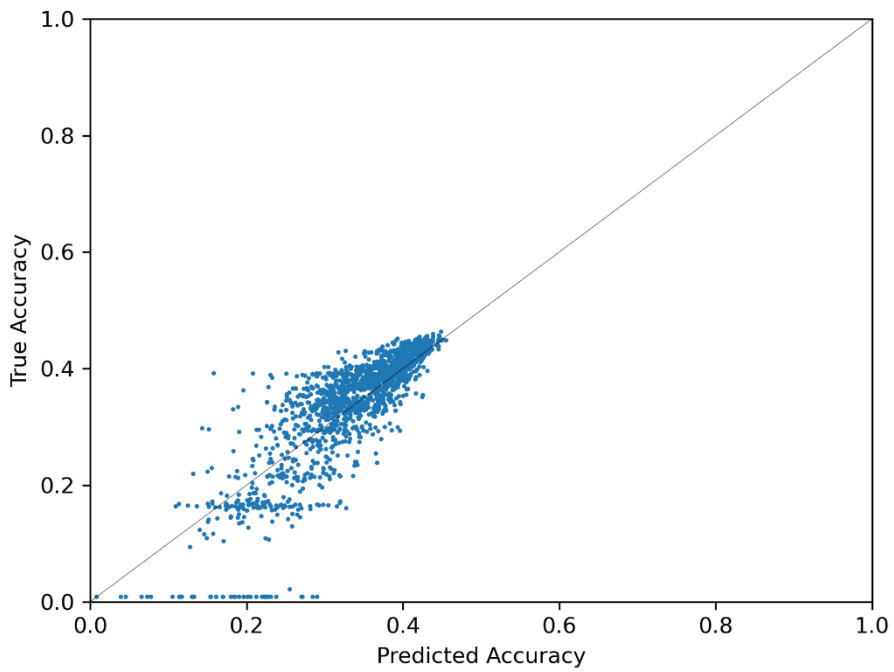
(c) Training set on CIFAR-100



(d) Training set on CIFAR-100



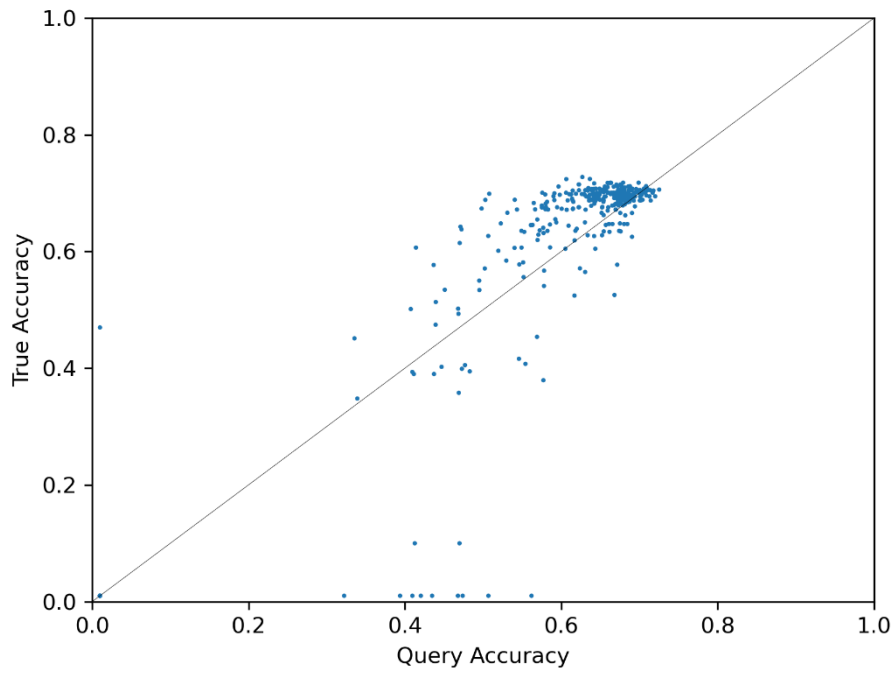
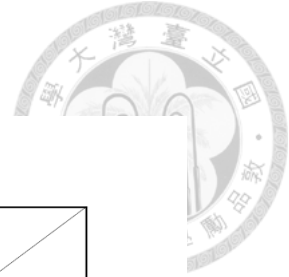
(e) Training set on ImageNet16-120



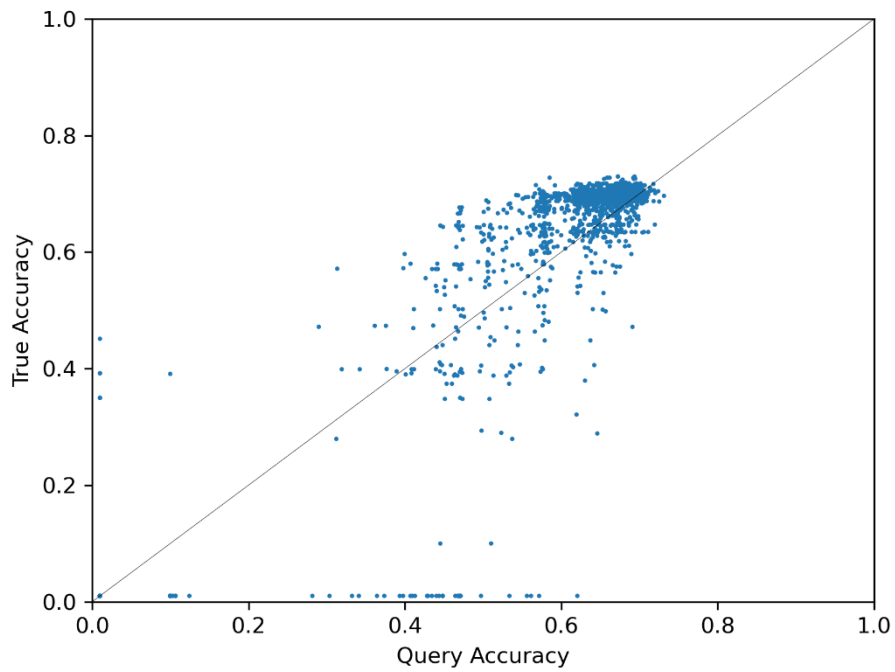
(f) Testing set on ImageNet16-120

Figure 15: Regression results on CIFAR-100 and ImageNet16-120.

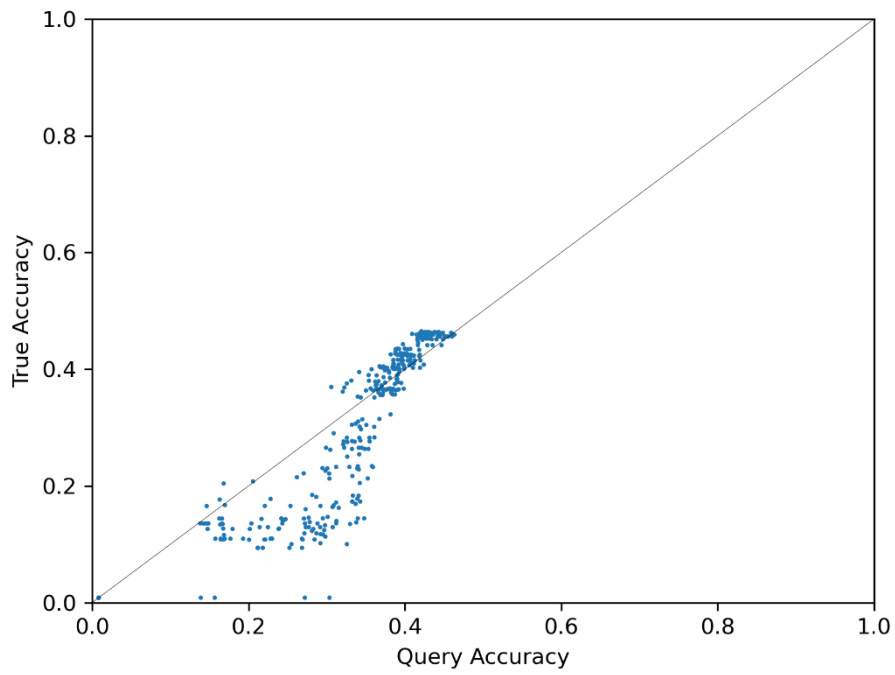
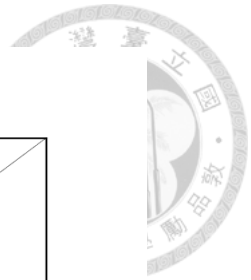
3. Inversion



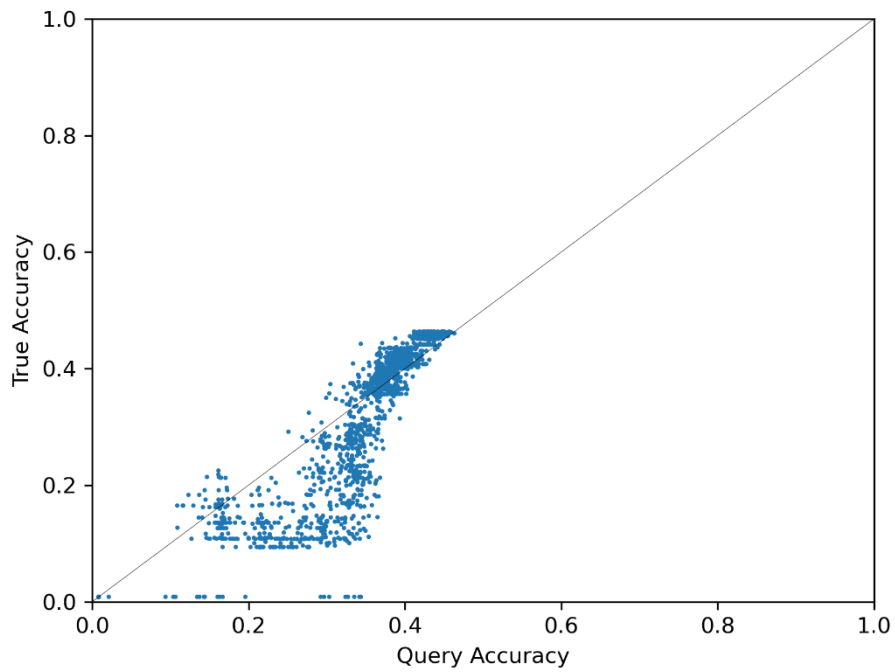
(a) Training set on CIFAR-100



(b) Testing set on CIFAR-100

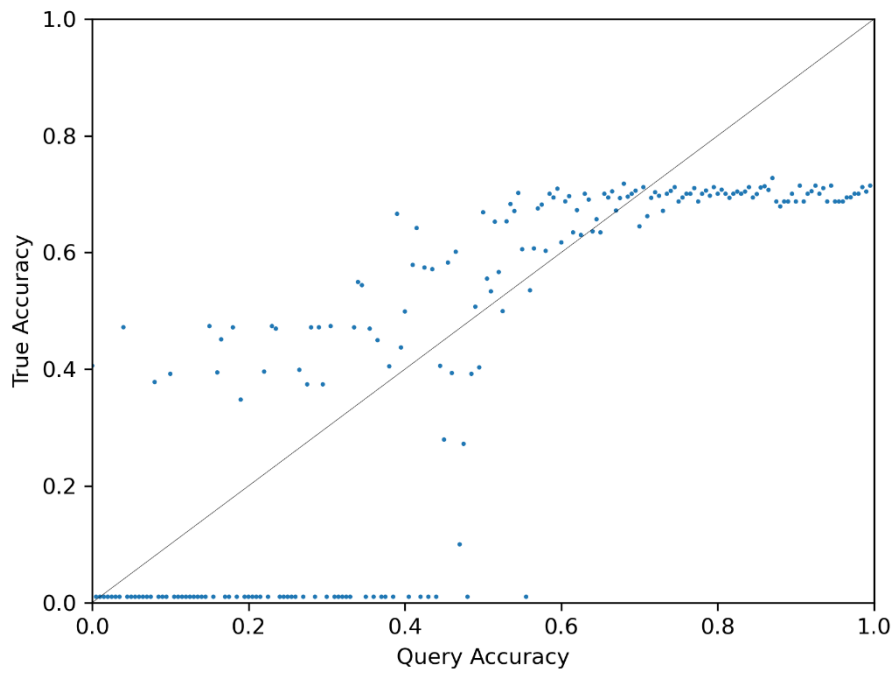


(c) Training set on ImageNet16-120

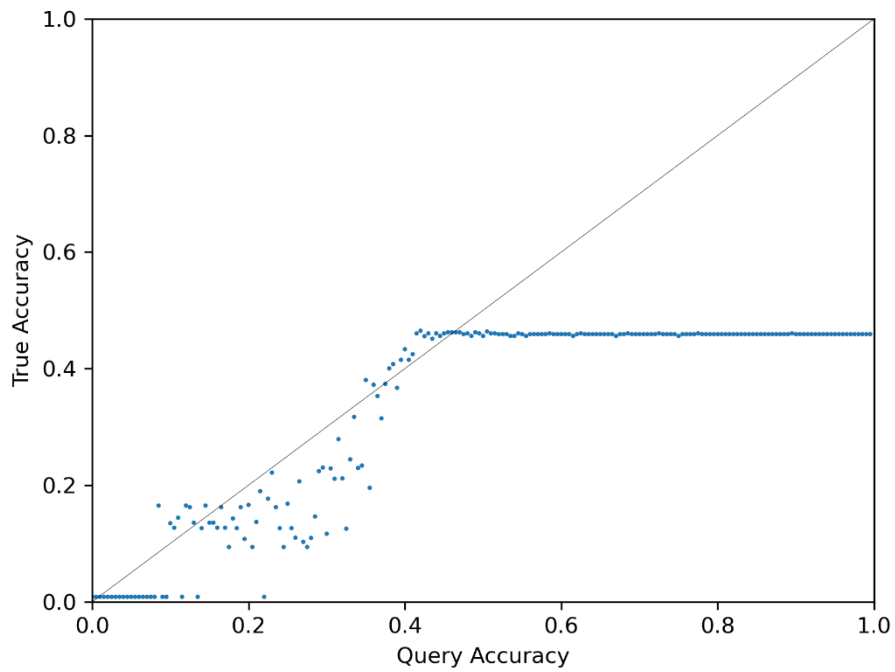


(d) Testing set on ImageNet16-120

Figure 16: Inversion results on CIFAR-100 and ImageNet16-120.



(a) Inversion result over a range on CIFAR-100.



(b) Inversion result over a range on ImageNet16-120.

Figure 17: Inversion results over a range.



B. Hyperparameters

Table 8: Hyperparameters of the GIN encoder.

Hyperparameter	Value
Latent Dimension	16
MLP Dimension	128, 128, 128, 128
MLP Activation	ReLU

Table 9: Hyperparameters of the Transformer decoder.

Hyperparameter	Value
Node Embedding (d_model)	32
Num Layer	3
Num Head	3
Feed Forward Dimension	256

Table 10: Hyperparameters of the pretraining.

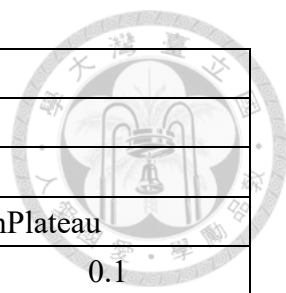
Hyperparameter	Value	
Optimizer	Adam	
Learning Rate	1e-3	
Batch Size	64	
Epoch	500	
Learning Rate Scheduler	ReduceLROnPlateau	
ReduceLROnPlateau	Factor	0.1
	Patience	50
	Min LR	1e-5
Early Stopping Patience	100	

Table 11: Hyperparameters of Invertible Neural Network.

Hyperparameter	Value
Couple Layer	4
Hidden Layer	4
Hidden Dimension	128

Table 12: Hyperparameters of fine-tuning

Hyperparameter	Value
Optimizer	Adam



Learning Rate	1e-3	
Batch Size	64	
Epoch	500	
Learning Rate Scheduler	ReduceLROnPlateau	
ReduceLROnPlateau	Factor	0.1
	Patience	25
	Min LR	1e-5
Early Stopping Patience	50	

Table 13: Hyperparameters of retraining

Hyperparameter	Value
Optimizer	Adam
Learning Rate	1e-3
Batch Size	64
Epoch	50
Early Stopping Patience	10

Table 14: Hyperparameters of searching

Hyperparameter	Value
Top-K	5
Max Round	100

