A APPENDIX

A.1 DATASET DETAILS

In the transductive learning task, node features and edges between nodes in the whole datasets were known beforehand. We learn from the already labeled training dataset and then predict the labels of the testing dataset. Three benchmark datasets were used in this setting: CiteSeerr (Giles et al., 1998), Cora (Kipf & Welling, 2016) and PubMed (Sen et al., 2008). All of the three benchmark datasets are citation networks. In citation networks, each node represents a work, and each edge shows the relationship between two papers in terms of citations. The datasets contain bag-of-words features for each node, and the goal is to categorize papers into various subjects based on the citation. In addition to the three benchmark datasets, we also employ another dataset Amazon Computers (McAuley et al., 2015). Amazon Computers is a subset of the Amazon co-purchase graph, where nodes represent commodities and edges connect them if they are frequently purchased together. Product reviews are encoded as bag-of-word feature vectors in node features, and class labels are assigned based on product category. The used datasets are concluded in Table 5.

Table 5: The Statistics of Datasets

	CiteSeer	Cora	PubMed	Computers
#nodes	2708	3327	19717	13752
#edges	5278	4552	44324	245861
#features	1433	3703	500	767
#classes	7	6	3	10

A.2 PROOF OF THEOREM 3.4 FOR THE DICTIONARY ORTHOGONALITY IN NAC

Theorem A.1. Let the neural weights of each operator in a deep neural network be an atom and stack them column-wisely, we can guarantee the existence of an orthogonal dictionary.

Proof. Given a dictionary $\boldsymbol{H} \in R^{n \times K}$, where n is the number of nodes and K is the number of opearators in each layer, we have its mutual coherence computed as follows,

$$\varphi = \max_{\boldsymbol{h}_i, \boldsymbol{h}_j \in \boldsymbol{H}, i \neq j} \left| \left\langle \frac{\boldsymbol{h}_i}{\|\boldsymbol{h}_i\|_2}, \frac{\boldsymbol{h}_j}{\|\boldsymbol{h}_j\|_2} \right\rangle \right|, \tag{8}$$

where $\varphi \in [0, 1]$, and $\langle \cdot \rangle$ denotes inner product. Here, each atom, $\boldsymbol{h}_i$, is the weights from an operator. The minimum of φ is 0 and is attained when there is an orthogonal dictionary, while the maximum is 1 and it attained when there are at least two collinear atoms (columns) in a dictionary.

Let $E_i = \frac{\boldsymbol{h}_i}{\|\boldsymbol{h}_i\|_2}$ and $E_j = \frac{\boldsymbol{h}_j}{\|\boldsymbol{h}_j\|_2}$, by Central Limit Theorem (Fischer, 2011), we know that $\langle E_i, E_j \rangle / \sqrt{n}$ converges to a normal distribution, i.e.,

$$\langle E_i, E_j \rangle = \lim_{n \to \infty} \sqrt{n} Z, \tag{9}$$

where Z is a standard normal distribution. Consider $\bar{E}$ as the mean value of all $\langle E_i, E_j \rangle$. With weak law of large numbers (a.k.a. Khinchin's law) (ter Haar, 1949), for any positive number ε, the probability that sample average $\bar{E}$ greater than ε converges 0 is written as

$$\lim_{n \to \infty} \Pr\left(\left| \bar{E} \right| \geq \varepsilon \right) = 0 \tag{10}$$

This implies that the probability that the inner product of E_i and E_j is greater than ε close to zero when $n \to \infty$. In other words, the probability that E_i and E_j are nearly orthogonal goes to 1 when their dimensionality is high. Therefore, the coherence of this dictionary reaches the minimum at a high dimensionality that holds for deep neural networks naturally. ☐

A.3 EXPERIMENTAL IMPLEMENTATION DETAILS

Environment. We implement experiments related to accuracy on Citeseer, Cora, and PubMed using PyTorch (Paszke et al., 2019) (version 1.10.2+cpu) on a CPU server that has a 48-core Intel Xeon

Platinum 8260L CPU, and experiments on Amazon Computers using PyTorch (version 1.10.2+gpu) on a GPU server with four NVIDIA 3090 GPUs (24G). Speed-related experiments on Citeseer, Cora, and PubMed were measured using PyTorch (version 1.4+cpu) on a CPU server with a ten-core Intel Xeon Platinum 8255C CPU, 40G RAM, and 500G DRAM. Speed-related experiments on Amazon Computers were measured using PyTorch (version 1.10.2+gpu) on a GPU server with four NVIDIA 3090 GPUs (24G). Operators used in the experiments are from the built-in functions of PyG (version 2.0.2) (Fey & Lenssen, 2019).

Searching configuration. In our experiments we adopt 3-layer GNN as the backbone. Unless specified, our experiments follow the same settings for searching architectures as SANE (Zhao et al., 2021b) :

- $Architecture\ optimizer$. We use Adam for training the architecture parameters $\boldsymbol{\alpha}$. We set the learning rate as 0.0003 and the weight decay as 0.001. Also, the β_1 and β_2 are fixed as 0.5 and 0.999, respectively. All runs a constant schedule for training, such as 100 epochs.
- $Weight\ optimizer$. We use SGD to update models' parameters, i.e., w. The learning rate and SGD momentum are given as 0.025 and 0, respectively, where the learning rate has a cosine decay schedule for 100 epochs. We fix the weight decay value, i.e. set $\rho_1 = 0.0005$.
- $Batch\ size$. For transductive tasks, we adopt in-memory datasets, and the $batch\_size$ is fixed as the size of the dataset themselves.

The configuration for retraining phase. At the retraining state, we adopt Adam as the optimizer and set the scheduler with cosine decay to adjust the learning rate. The total number of epochs is fixed 400 for all methods for fairness. Please refer to the setting of SANE (Zhao et al., 2021b) and EGAN (Zhao et al., 2021a) for more details as we follow this in our experiment.

For CiteSeer dataset, we set the initial learning rate as 0.005937 and weight decay as 0.00002007. The configuration for models is as follows: $hidden\_size = 512$, $dropout = 0.5$, and using $ReLU$ as the activation function.

For Cora dataset, we set the initial learning rate as 0.0004150, and weight decay as 0.0001125. In model, we set $hidden\_size = 256$, $dropout = 0.6$, and use $ReLU$ as the activation function.

For PubMed dataset, we set the initial learning rate as 0.002408 and weight decay as 0.00008850. As for the model, we have $hidden\_size = 64$ and $dropout = 0.5$, and use $ReLU$ as the activation function.

For Amazon dataset, we set the initial learning rate as 0.002111 and weight decay as 0.000331. As for the model, we have $hidden\_size = 64$ and $dropout = 0.5$, and use elu as the activation function.

Solving L^1 regularization. The L^1 regularization, also known as **Lasso Regression** (Least Absolute Shrinkage and Selection Operator), adds an absolute value of the magnitude of the coefficient as a penalty term to the loss function (Ranstam & Cook, 2018). Using the L^1 regularization, the coefficient of the less important feature is usually decreased to zero, sparsifying the parameters. It should be noted that since $||\boldsymbol{\alpha}||_1$ is not differentiable at $\boldsymbol{\alpha} = \mathbf{0}$, the standard gradient descent approach cannot be used.

Despite the fact that the loss function of the Lasso Regression cannot be differentiated, many approaches to problems of this kind, such as (Schmidt et al., 2009), have been proposed in the literature. These methods can be broadly divided into three groups: constrained optimization methods, unconstrained approximations, and sub-gradient methods.

Since subgradient methods are a natural generalization of gradient descent, this type of methods can be easily implemented in Pytorch's framework. Lasso Regression can be solved using a variety of subgradient techniques; details on their implementation can be found in (Fu, 1998) and (Shevade & Keerthi, 2003).

Computational Complexity Estimation of NAC. The computation of NAC has two major parts: the forward pass and the backward pass. Given the search space, the computation of the forward is then fixed and regarded as a constant. Therefore, the computational complexity mainly focuses on the backward pass in the NAC algorithms.

The main version of our work does not need to update weights, but only to update architectural parameter $\boldsymbol{\alpha}$ during the training process. Therefore, the algorithmic complexity is as $O(T * \|\boldsymbol{\alpha}\|)$, which is a *linear* function w.r.t $\boldsymbol{\alpha}$. The dimension of $\boldsymbol{\alpha}$ is often small, which makes the model easy

to scale to large datasets and high search space. When updating weights of the linear layer, the complexity is estimated as $O(T * (\|\boldsymbol{\alpha}\| + \|\boldsymbol{W}_o\|))$. The dimension of $\boldsymbol{W}_o$ is a constant number, that equals the number of classes. Therefore, the complexity is almost the same as the main version of NAC, where the complexity is $O(T * \|\boldsymbol{\alpha}\| + \|\boldsymbol{W}_o\|)$.
When updating weights, similar to DARTS, the complexity is estimated as $O(T * (\|\boldsymbol{\alpha}\| + \|\boldsymbol{w}\|))$. The dimension of $\boldsymbol{w}$ is often much larger than $\boldsymbol{\alpha}$, therefore, the complexity is dominated by updating $\boldsymbol{w}$, where the complexity is $O(T * \|\boldsymbol{w}\|)$. Since the dimension of $\boldsymbol{\alpha}$ is much smaller than $\boldsymbol{w}$, the complexity of NAC is much less than this type of methods.

Approximate Architecture Gradient. Our proposed theorems imply an optimization problem with $\boldsymbol{\alpha}$ as the upper-level variable and $\boldsymbol{W}_o$ as the lower-level variable:

$$\begin{cases} \boldsymbol{\alpha}^* = \underset{\boldsymbol{\alpha}}{\operatorname{argmax}} \mathcal{M}\left(\boldsymbol{W}_o^*(\boldsymbol{\alpha}), \boldsymbol{\alpha}\right) \\ \boldsymbol{W}_o^*(\boldsymbol{\alpha}) = \underset{\boldsymbol{W}_o}{\operatorname{argmin}} \mathcal{L}(\boldsymbol{\alpha}, \boldsymbol{W}_o), \end{cases} \tag{11}$$

Following (Liu et al., 2019a), we can adopt a First-order Approximation to avoid the the expensive inner optimization, which allows us to give the implementation in the algorithm 1.

A.4 ABLATION STUDIES

A.4.1 THE EFFECT OF SPARSITY

Our model uses the hyperparameter ρ to control the sparsity of the architecture parameter $\boldsymbol{\alpha}$, where a large sparsity is to enforce more elements to be zero. We investigate the effect of this hyperparameter by varying its value in a considerably large range, such as $[0.001, 10]$. We first present the accuracy of different sparse setting in Fig. 4. We find that the results vary little in a considerably wide range, this indicates our models are insensitive to sparsity hyperparameter in general.

A.4.2 THE EFFECT OF RANDOM SEEDS

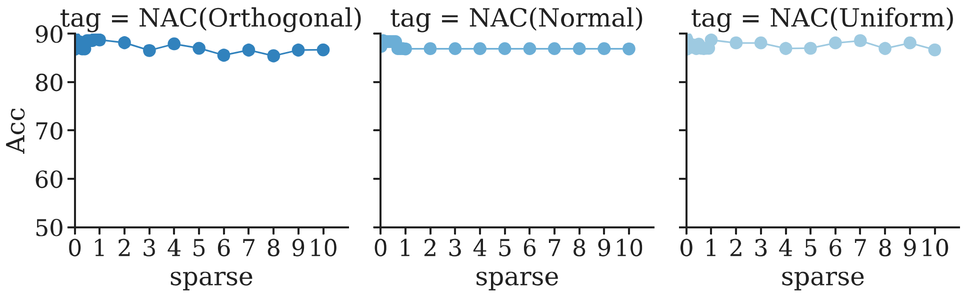
Random seeds often play an importance role in traditional NAS methods as it affects the initialization significantly. People often report the average or the best results under different random seeds, this may lead to poor reproducibility. To the best our knowledge, this is for the first we explicitly demonstrate the effect of random seeds in this subject. We run experiments on several random seeds and report the results of NAC on Pubmed dataset, as shown in Fig. 5. In particular, we implement multiple combinations of random seeds and sparsity to observe the variation on performance. Note that we round the values to integer to fit the table. In all these combinations, we have the average and variance as 87.32% and 0.9%, respectively. The average performance is comparable to the best results from all competitive results, which indicates the stability of NAC.

A.4.3 THE EFFECT OF TRAINING ON THE FINAL LINEAR LAYER

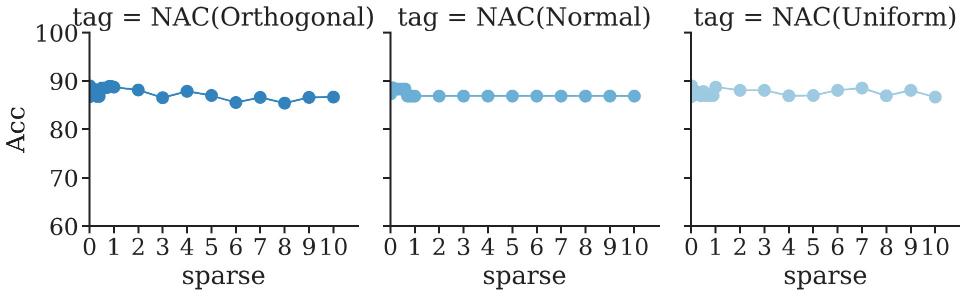
Our proposed theorems prove that a GNN with randomly initialized weights can make the final output as good as a well-trained network when initializing networks with orthogonal weights and updating the total network using gradient descent. In practice, we find it difficult to determine at what training epoch the optimal weight parameters can be obtained through training linear layer. We noticed that most of the time, the untrained weights in the initial state can often already exceed the accuracy that can be obtained from the weights after multiple epochs of training the final linear layer, as shown in Fig. 6. Therefore, we further omit the training of the final linear layer. It is important to note that this approximation is based on our proposed theorems in which most of the intermediate layers do not require training.

A.5 RUNTIME OF EACH METHOD ON A SINGLE GPU SERVER

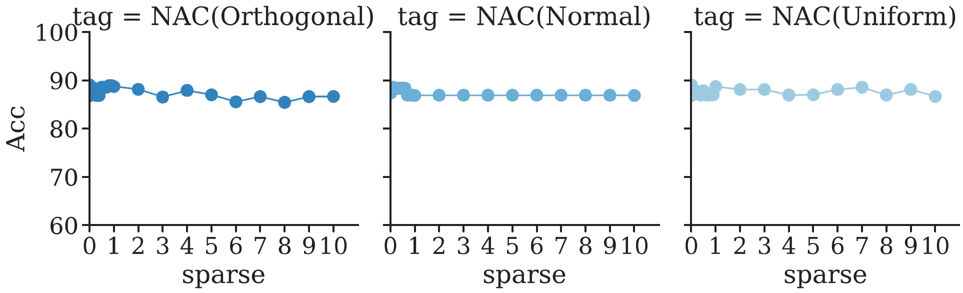
Apart from the running time on the CPU, we also measure the running time for all methods on a GPU platform, where we use PyTorch (version 1.10.2+gpu) on a GPU server with four NVIDIA 3090 GPUs (24G), as shown in Table 6. These results are consistent with the ones in Table 1, demonstrating our advantage in speed.



(a) CiteSeer



(b) Cora



(c) PubMed

Figure 4: Sensitivity study of the sparsity. Results are with varying sparsity (x-axis) and different initialization NAC methods (i.e., normal, uniform and orthogonal). The variation is small, showing the robustness of our model w.r.t the sparsity.

Table 6: Timing results of the compared methods and our NAC method in one the same GPU server. NAC attains superior performance in efficiency (in seconds).

	CiteSeer	Cora	PubMed	Computers
RS	196.00	328.00	461.00	900.00
BO	225.00	355.00	462.00	878.00
GraphNAS	6193.00	6207.00	6553.00	8969.00
GraphNAS-WS	947.00	1741.00	2325.00	4343.00
SANE	35.00	41.00	43.00	43.00
NAC	**14.00**	**14.00**	**15.00**	**14.00**
NAC-updating	42.00	31.00	36.00	42.00

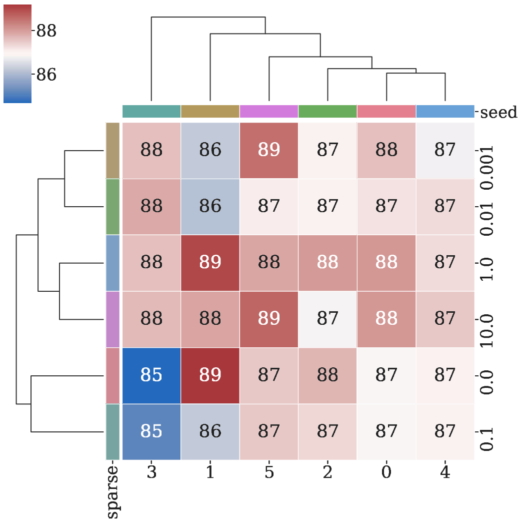


Figure 5: The effects of random seeds of NAC on the accuracy, where x-axis denotes the random seeds and y-axis denotes the sparsity. NAC performs stably with random seeds.

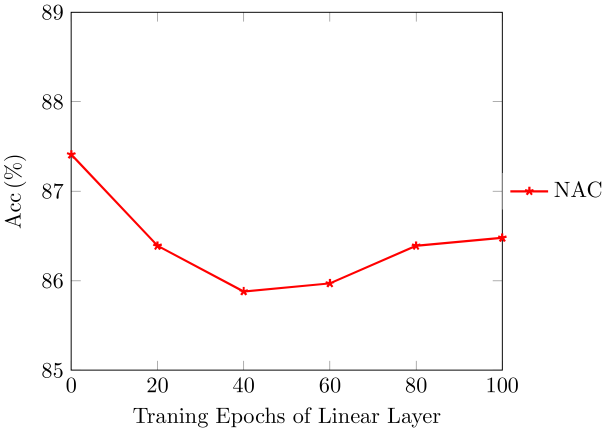


Figure 6: The effects of training final linear layer of NAC on the accuracy, where x-axis denotes the training epochs of the final linear layer and y-axis denotes the averaged accuracy of acquired architecture α using the corresponding weights.