INTRA-LAYER NEURAL ARCHITECTURE SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

We propose an efficient neural architecture search (NAS) algorithm with a flexible search space that encompasses layer operations down to individual weights. This work addresses NAS challenges in a search space of weight connections within layers, specifically the large number of architecture variations compared to a high-level search space with predetermined layer types. Our algorithm continuously evolves network architecture by adding new candidate parameters (weights and biases) using a first-order estimation based on their gradients at $\mathbf{0}$. Training is decoupled into alternating steps: adjusting network weights holding architecture constant, and adjusting network architecture holding weights constant. We explore additional applications by extend this method for multi-task learning with shared parameters. On the CIFAR-10 dataset, our evolved network achieves an accuracy of 97.42% with 5M parameters, and 93.75% with 500K parameters. On the ImageNet dataset, we achieve 76.6% top-1 and 92.5% top-5 accuracy with a search restriction of 8.5M parameters.

1 INTRODUCTION

Deep neural networks have very been successful in tackling various learning problems in computer vision provided large amounts of training data LeCun et al. (2015). However, designing neural network architectures for a particular task, especially under resource constraints, is a challenge that requires some insights on the task and likely the involvement of a human expert. Researchers have spent much effort attempting to automate this process, proposing a variety of methods known collectively as neural architecture search (NAS) algorithms Jaafra et al. (2018). These methods use a wide variety of techniques such as random search Li & Talwalkar (2019), reinforcement learning Zoph & Le (2017), tournament selection Real et al. (2017), meta-networks Luo et al. (2018), etc. to determine the layer types and connections in the network. At a high level, a NAS algorithm defines a search space $\mathcal{A}$ of architectures usually with a range of possible layer types and activation functions. Using the aforementioned methods it chooses an architecture $A \in \mathcal{A}$, evaluates its performance, then continues the search by attempting a new choice A' that is expected to improve upon the current A. These algorithms generally treat architecture search as hyper-parameter optimization problem and earlier works focused largely on the accuracy of the architecture. However, as neural networks are used widely today from data centers to mobile devices, the memory and computational cost of inference is an increasingly important factor. Therefore, we place additional emphasis on practical issues such as training efficiency and network size.

While most past works search for network architectures at the granularity of layers and layer-level connections Elsken et al. (2018), this paper addresses NAS at the level of individual parameters. We propose intra-layer neural architecture search (ILNAS), a training process that integrates architecture search with weight training, in essence, performing gradient descent on both weight values and connections. Our search space consists of convolution and reduction cells similar to NASNet Zoph et al. (2018), but allows for architecture optimization within layers. This means that linear layers, theoretically unbounded in width and depth, can be arbitrarily connected and convolutional layers can be irregular with varying kernel dimensions, missing kernel weights (e.g. atrous), etc. The process can be considered as reverse network pruning: we continuously expand a baseline network by adding new weights rather than deleting them from a large pre-trained network.

The training process consists of two interleaved tasks: **1.** Given the current network architecture, we train by adjusting the current weights. **2.** Given the current weights, we train by adjusting the current architecture. To evaluate our method, we use the popular image classifications datasets

ImageNet Russakovsky et al. (2015), CIFAR-10 Krizhevsky (2009), STL-10 Coates et al. (2011), Fashion-MNIST Xiao et al. (2017), and MNIST LeCun & Cortes (2010). We compare test accuracies achieved under parameter constraints against NAS algorithms in other recent works. We also evaluate transfer and multi-task learning with the learned architectures to classify different or multiple datasets with continuous evolution. Our results show that network evolution achieves competitive results compared to state-of-the-art NAS approaches in terms of both accuracy and resources. In addition, we show that this algorithm can be extended to learning multiple tasks by training multiple architectures with parameter sharing.

2 Related Work

Research in neural architecture search dates back a few decades to genetic algorithms that were proposed to find both architecture and weights Schaffer et al. (1992). One example is the NEAT algorithm Stanley & Miikkulainen (2002) which trains a population of models and uses a survival of the fittest strategy to continuously improve. Evolutionary algorithm (EA) have seen a resurgence in interest and recent works such as Real et al. (2017) have demonstrated success in finding architectures for image classification. In most cases, however, EA methods concurrently train multiple models and require significant computation resources.

Another class of methods use reinforcement learning (RL). With RL, one can model parameter choices for each layer of the network as a Markov decision process (MDP). In this environment, an agent goes through a phase of exploration where it is encouraged to choose previously untested parameters for each layer, and a phase of exploitation where it chooses the best parameters based on past results. Baker et al. proposed MetaQNN Baker et al. (2017), a meta-algorithm that generates CNN architectures using Q-learning in this manner. The main trade-off in these greedy approaches is that over-exploration is costly and leads to slow convergence whereas under-exploration potentially converges the network to poor local minima. Hypernetworks (or meta-networks) use a separate neural network to output a description of the architecture or make predictions on the performance of an architecture. Luo et. al. Luo et al. (2018) use an encoder to map the architecture into a continuous space where gradient ascent can be performed based on estimated accuracy. A decoder then retrieves the architecture from this space. Other recent approaches include using recurrent neural network (RNN) controllers to generate layer parameters Zoph & Le (2017); Zoph et al. (2018) and graph methods that use various mutations on the computation graph to generate new architectures Liu et al. (2018). If training efficiency is of greater concern, one-shot training methods that train the network only once have been proposed. They are often combined with one or more of the above methods with additional optimizations such as weight sharing to train the network with relatively little overhead compared to regular training Pham et al. (2018).

This work differs most notably from existing approaches in its search space that consists of individual weight connections and kernels between layers rather than the network's global or cell structure. As a result, the discovered architecture can have irregular linear layers that are neither fully convolutional nor fully connected. This allows fine grained control over the exact size of the trained network (i.e. the number of weights and multiply-accumulate operations) but also presents challenges due to the enormous size of the search space. Any NAS method based on evaluating each new architecture candidate is not scalable down to individual weights unless a tremendous amount of computation resource is available. Finally, since our search space does not fully overlap with those of many related approaches, our method can be used in conjunction with other NAS algorithms that search for different aspects of network architecture (e.g. activation functions, architecture template, cell structure).

3 Architecture Evolution

We consider a typical classification problem with a real-valued input vector $\boldsymbol{x}$ and labels y. A feedforward neural network with linear layers, activation function $\sigma(x)$, and flattened weight vector $\boldsymbol{W} \in \mathbb{R}^d$ defines a function $f : (\boldsymbol{x}, \boldsymbol{W}) \mapsto y$ as follows:

$$f(\boldsymbol{x}, \boldsymbol{W}) = \text{argmax}(...\boldsymbol{W}^{(3)}\sigma(\boldsymbol{W}^{(2)}\sigma(\boldsymbol{W}^{(1)}\boldsymbol{x}))...).$$

Generally, some variant of gradient descent is used in an attempt to find loss-minimizing weights $\boldsymbol{W}$ for the network. The focus of this paper is to optimize network architecture in addition to weights.

3.1 PROBLEM FORMULATION

We define the network's architecture A to be the vector $\boldsymbol{\alpha}$ with the same dimension as $\boldsymbol{W}$ whose elements $\alpha_i \in \{0, 1\}$ are Boolean. A network f with architecture $\boldsymbol{\alpha}$ is then expressed as follows:

$$f : (\boldsymbol{x}, \boldsymbol{\alpha} \odot \boldsymbol{W}) \mapsto y, \;\; \boldsymbol{W} \in \mathbb{R}^d, \boldsymbol{\alpha} \in \mathbb{B}^d.$$

This definition is used identically in network pruning, however the dimensions of $\boldsymbol{\alpha}$ and $\boldsymbol{W}$ are dynamic during training. Viewing the network as a graph, a 1 element in the architecture vector $\boldsymbol{\alpha}$ indicates the presence of a corresponding weight or bias term in $\boldsymbol{W}$, and a 0 element indicates the absence of a term. Hence, a Boolean vector in $\mathbb{B}^d$ maps to a subset of functions of $f_{\boldsymbol{W}}(\boldsymbol{x})$ with certain weights zeroed. Network architecture search for training inputs $\boldsymbol{X}$ and labels $\boldsymbol{Y}$ under some parameter constraint $|\boldsymbol{\alpha}| \leq c$ can be formulated as a bilevel optimization problem:

$$\min_{|\boldsymbol{\alpha}| \leq c} L(f(\boldsymbol{X}, \boldsymbol{w}(\boldsymbol{\alpha})), \boldsymbol{Y}), \tag{1}$$

$$\text{where } \boldsymbol{w}(\boldsymbol{\alpha}) = \underset{\boldsymbol{W}}{\operatorname{argmin}}\, L(f(\boldsymbol{X}, \boldsymbol{\alpha} \odot \boldsymbol{W}), \boldsymbol{Y}). \tag{2}$$

This formulation is similar to DARTS Liu et al. (2019) and SNAS Xie et al. (2019) however $\boldsymbol{\alpha}$ is generalized to encode any subset of network weights rather than layer-level operations. This optimization problem is difficult for two reasons: **1.** the upper level decision space is large and consists of non-differentiable Boolean values, and **2.** the lower level optimization for $\boldsymbol{W}$ is computationally costly if we train every architecture $\boldsymbol{\alpha}$.

3.2 ADDING NEW PARAMETERS

Before describing the optimization algorithm, we visually demonstrate how new parameters expand the network. Figure 1 shows three types of architectural additions that do not initially affect the network. Nodes in this figure $N_{00}, N_{01}, ...$ represent a single neuron with one output value. A new connection between existing nodes (shown left) is the simplest. The missing connection is simply treated as a 0 weight as defined above ($\boldsymbol{\alpha} \odot \boldsymbol{W}$ with $\boldsymbol{\alpha} = [..., 0, ...]$). However, we can still compute loss gradient on the weights since $\partial \boldsymbol{W}\boldsymbol{x}/\partial \boldsymbol{W}$ does not depend on the current $\boldsymbol{W}$. Adding a new node (shown center) is done by forming a connection from a placeholder node to the next layer. A placeholder node here indicates one without forward connections and hence has no impact on the network's output. As before, zeroed weights for forward connections are differentiable. These nodes are candidates to be selected and added to the network during training.

A useful property of ReLU activations $\sigma(x) = \texttt{max}\{0, x\}$ is that it is unchanged by composition, i.e. $\sigma(x) = (\sigma \circ \sigma \circ \sigma ...)(x)$. Each node can be extended into a composition of as many nodes as needed with all connections weighted at 1 (and bias at 0) without changing its current input-output mapping.

$$\sigma(x) = ...\sigma(w\sigma(w\sigma(x) + b) + b)... \quad (w = 1,\, b = 0).$$

As shown on the right of Figure 1, we can exploit this property to add an arbitrary number of layers to the network without drawback, hence the network is unbounded in depth.

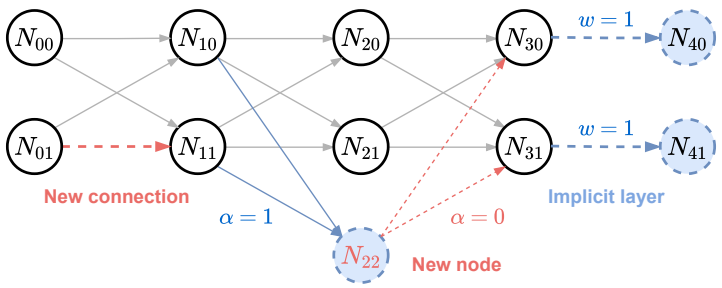


Figure 1: Left - adding a new weight connection from N_{01} to N_{11}. Center - adding a placeholder node N_{22} to the network by adding connections to N_{30} and N_{31}. Right - forming a new layer with N_{40}, N_{44} without affecting the network's outputs.

3.3 ARCHITECTURE OPTIMIZATION

We use a greedy algorithm, at each step, given the current architecture $\boldsymbol{\alpha}$, our objective is to determine the best way to expand it $\boldsymbol{\alpha} \rightarrow \boldsymbol{\alpha'}$ such that $\boldsymbol{w}(\boldsymbol{\alpha'})$ minimizes empirical risk. We use a two step process to select new parameters to be added to the current network architecture. Prior to an architecture optimization step, we assume the current network is trained to some local minima, thus its weight gradient is non-zero only for non-existent weights in the current architecture. For example, if the first j weights are active (i.e. $\alpha_1, ... \alpha_j = 1$, $\alpha_{j+1}, \alpha_{j+2}, ... = 0$), then we have:

$$\nabla_W L(f(\boldsymbol{X}, \boldsymbol{\alpha} \odot \boldsymbol{W}), \boldsymbol{Y}) = \nabla_W L(\boldsymbol{\alpha}, \boldsymbol{W}) = [0, 0, 0, ... \partial L/\partial W_{j+1}, \partial L/\partial W_{j+2}, ...].$$

We choose first level candidate parameters based on the magnitude of the derivative of missing connections: $|\partial L/\partial(\alpha_i W_i)|$ where $\alpha_i = 0$. We noted that since layers are linear, partial derivatives with respect to zero weights are not necessarily zero. Naturally, the larger this value, the larger the initial decrease in loss we expect if W_i becomes a non-static network weight. In this step, we denote the new 'hyperarchitecture' constructed by adding up to n candidate parameters included as $\boldsymbol{\alpha^*}, |\boldsymbol{\alpha^*}| - |\boldsymbol{\alpha}| = n$ and updating all weights $\boldsymbol{W^*}$ for this gradient descent step. These are the largest n parameters based on their partial derivative magnitudes. From here, we want to determine the m best parameters to add from candidates ($m \leq n$) such that the new architecture $\boldsymbol{\alpha'}$ that minimizes loss after weight training.

$$\min_{|\boldsymbol{\alpha'}| \leq |\boldsymbol{\alpha}| + m} L(\boldsymbol{\alpha'}, \boldsymbol{W'}), \text{where } \boldsymbol{W'} = \underset{\boldsymbol{W}}{\operatorname{argmin}}\, L(f(\boldsymbol{X}, \boldsymbol{\alpha'} \odot \boldsymbol{W^*}), \boldsymbol{Y}).$$

The variables n and m change during training and are determined from on the architecture learning rate A_{lr} that specifies how new parameters should be added each expansion step given the parameter constraints of the search. For large networks with millions of parameters, the number of first level candidates is already large. Evaluating $\binom{n}{m}$ possible architecture choices is impractical, especially since inner optimization on weights requires additional training. Indeed, if the magnitude of the Hessian is small, a linear approximation of weight changes across training iterations can be obtained solely from the current weight gradients, so choosing the m largest among them like we did for $\boldsymbol{\alpha^*}$ above should suffice. However, we perform an additional iteration of gradient descent on weights for the hyperarchitecture $\boldsymbol{\alpha^*}$ to obtain another reference point. We simply linearly combine the gradients of the base $\boldsymbol{\alpha}$ and the hyperarchitecture $\boldsymbol{\alpha^*}$ to obtain a final magnitude $|\partial L/\partial(\alpha_i W_i) + \partial L/\partial(\alpha_i^* W_i^*)|$ to sort by the largest m. Our architecture expansion decisions can be rapidly determined from two iterations of training.

3.4 SEARCH SPACE GRANULARITY

If each element of $\boldsymbol{\alpha}$ is an independent Boolean variable, the search space would be at its largest. On the other hand, we can group multiple element of $\boldsymbol{\alpha}$ as one variable to control the search space granularity. Figure 2 shows various levels of architecture search space from layer-level connections to individual weights.

Optimization is more costly and difficult the more fine grained the search space, hence we use a top-down approach and start with architecture changes at the level of layers and kernels. We use an architecture template based on NASNet described in Section 4.1. In our experiments we do not search at the level of layers to isolate the effectiveness of ILNAS, we search only at the level of kernels and individual weights.

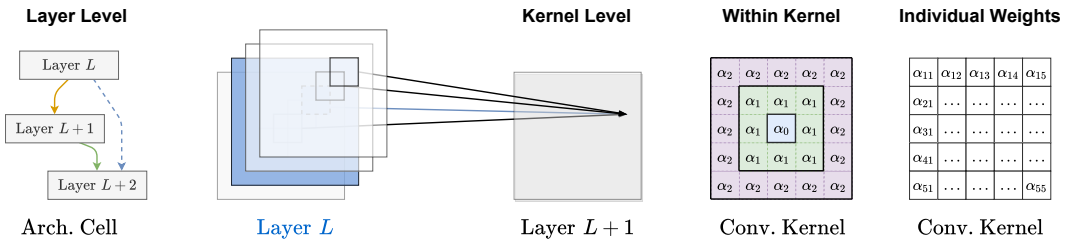


Figure 2: Dynamic search space granularity that varies from layer-level operations down to individual weights and biases.

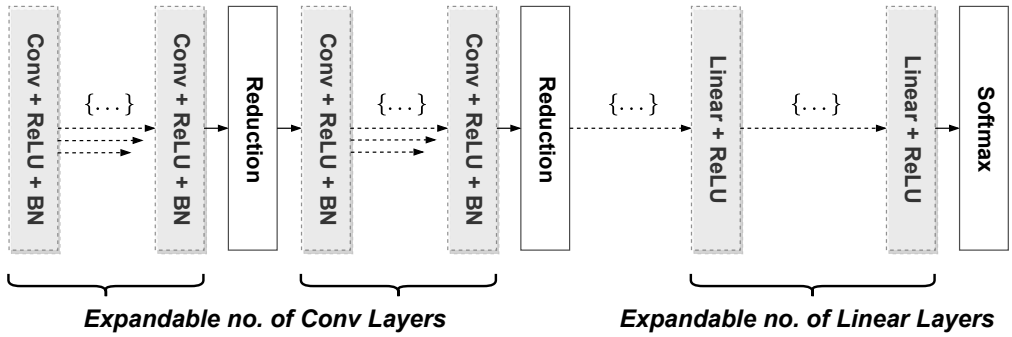


Figure 3: Network architecture template consisting of variable convolutional layers between reductions followed by variable linear layers.

4 TRAINING METHODS

We separate the training process into two independently executed tasks where one variable is isolated in each. As training samples are evaluated, we perform the following:

1. Given the current architecture $\boldsymbol{\alpha}$, adjust the weights $\boldsymbol{W} \mapsto \boldsymbol{W}'$ such that the network $f(\boldsymbol{x}, \boldsymbol{\alpha} \odot \boldsymbol{W}')$ can achieve lower loss.
2. Given the current weights $\boldsymbol{W}$, adjust the architecture $\boldsymbol{\alpha} \mapsto \boldsymbol{\alpha}'$ such that the network $f(\boldsymbol{x}, \boldsymbol{\alpha}' \odot \boldsymbol{W})$ can achieve lower loss.

The algorithm pseudocode is given in Algorithm 1 and consists of the two phases we described: Phase I: standard network weight training, and Phase II: network architecture expansion. The function `AddCandidates` add n non-architectural weights with the largest gradient magnitudes to the network. The function `UpdateArch` then selects a subset of m parameters to add to the network based on the base and hyper-architecture as described in the previous section. Based on the parameter constraints and number of training epochs, we can determine how many parameters are added each epoch.

Algorithm 1 ILNAS Training Algorithm

`Net` $\leftarrow$ Base network with randomly initialized architecture.
while $n \leq$ numEpochs **do**
  1. Perform standard SGD by updating weights based on $\nabla_W L(\boldsymbol{\alpha}, \boldsymbol{W})$.
  if $\nabla_W L(\boldsymbol{\alpha}, \boldsymbol{W}) \approx \mathbf{0}$ or j epochs elapsed **then**
    2.1 Compute network gradient $\nabla_W L(\boldsymbol{\alpha}, \boldsymbol{W})$ and add them to the hypernetwork.
    `HyperNet` $\leftarrow$ `Net.AddCandidates`$(\nabla_W L(\boldsymbol{\alpha}, \boldsymbol{W}))$
    2.2 Compute hypernet gradient $\nabla_W L(\boldsymbol{\alpha}^*, \boldsymbol{W}^*)$ and determine the new architecture $\boldsymbol{\alpha}'$.
    `Net` $\leftarrow$ `Net.UpdateArch`$(\boldsymbol{\alpha}, \boldsymbol{\alpha}^*, \nabla_W L(\boldsymbol{\alpha}, \boldsymbol{W}), \nabla_W L(\boldsymbol{\alpha}^*, \boldsymbol{W}^*))$
  end if
end while

4.1 ARCHITECTURE TEMPLATE

We use an architecture template similar to NASNet Zoph et al. (2018) as shown in Figure 3. The network consists of dimension preserving convolution layers with reduction layers inserted between them. Note that we do not define layer types since they are all custom. Every layer between two reduction cells can be connected to previous layers, and the total number of layers is unbounded (but limited in implementation). At the start of training, we randomly sample a baseline architecture to evolve from.

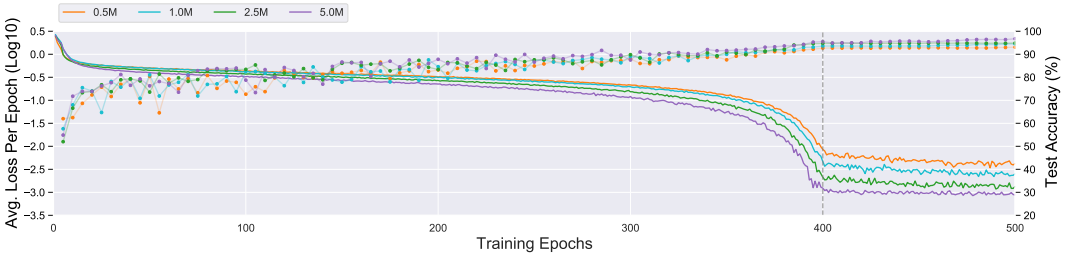


Figure 4: Empirical loss and test accuracy results after 500 epochs of training for evolutionary networks constrained to 0.5, 1.0, 2.0, and 5.0 million parameters.

4.2 Extending to Multi-task Learning

We apply our algorithm to train one neural network for more multiple tasks with some of its parameters shared and others independent across tasks. For example, in the case of lifelong learning, the network is expected to learn multiple tasks arriving in sequence. Thus, we extend our algorithm to consider multiple architectures $\boldsymbol{\alpha_1}, \boldsymbol{\alpha_2}, \boldsymbol{\alpha_3}, ...$ applied on the same network f depending on which task is being performed, the resulting networks are $f(\boldsymbol{x}, \boldsymbol{\alpha_1} \odot \boldsymbol{W}), f(\boldsymbol{x}, \boldsymbol{\alpha_2} \odot \boldsymbol{W}), f(\boldsymbol{x}, \boldsymbol{\alpha_3} \odot \boldsymbol{W}), ...$ with weights $\boldsymbol{W}$ partially shared. We apply the same NAS algorithm for each network architecture, however shared weights have gradients that are linearly combined for each architecture, we write: $\nabla_W f = \nabla_W f(\boldsymbol{\alpha_1} \odot \boldsymbol{W}) + \nabla_W f(\boldsymbol{\alpha_2} \odot \boldsymbol{W}) + ...$ for all architectures $\boldsymbol{\alpha_1}, \boldsymbol{\alpha_2}, ...$ We apply the ILNAS algorithm to expand each architecture individually and the shared portion as a standalone architecture.

5 Experiments

We implement our training algorithm in PyTorch and evaluate it on the popular image classification datasets MNIST LeCun & Cortes (2010), Fashion MNIST Xiao et al. (2017), CIFAR-10 Krizhevsky (2009), STL-10 Coates et al. (2011), and ImageNet Russakovsky et al. (2015). Initial weights are randomized by PyTorch defaults, and cross-entropy loss is used for all datasets. A hard limit of maximum 1024 channels and 64 convolutional layers deep is imposed on the network. Our hardware training platform consists of a single Nvidia RTX 2060 Super GPU and Intel Xeon E5-2660 v4 CPU (used for architecture expansion functions).

5.1 Image Classification

CIFAR-10. The CIFAR-10 Krizhevsky (2009) dataset offers 50,000 training images and 10,000 test images of dimension 32×32 for classification of simple objects into 10 categories. We use a batch size of 128 and perform standard pre-processing with input normalization, random crops, and horizontal flips. Four networks are trained for 500 epochs, each with a different parameter restriction of 0.5, 1.0, 2.5 and 5.0 million respectively. We perform mixed architecture and weight training in the first 400 epochs but fix the architecture for the last 100 epochs.

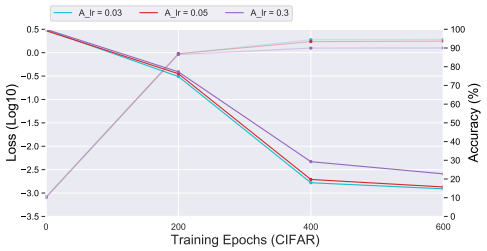
Figure 4 shows training losses of all four networks at every epoch and test accuracies every five epochs. A clear inverse correlation between network size and empirical loss is as expected. The network allowed to evolve to 5M parameters achieves a higher accuracy of 97.42% compared to the 93.75% accuracy of the smallest network with 500K weights. Figure 5 shows training results compared to a fixed network architecture and randomly evolved architecture of the same size (1.0M parameters, best of 5 trained). ILNAS clearly outperforms a random baseline and a fixed identically sized network. The architecture learning rate A_{lr} regulates how many new parameters are added in each expansion relative to the current network size. Its value decays as the epoch number increases. Under the same total parameter constraint, we can have few expansions with many new parameters added each step (large A_{lr}), or many expansions with few additions. The results show that, as expected, the latter performs better at the cost of increased training time.

Table 1 compares our results in CIFAR-10 with other recent NAS methods. Our training approach yields results competitive with state of the art RL and EA-based methods despite its significantly

Work	Method Type	Accuracy (%)	Params (M)	GPU Days
Baker et al. (2017)	RL	93.08	11.2	100
Zoph & Le (2017)	RL	96.35	37.4	22400
Zoph et al. (2018)	RL	96.59	3.3	2000
Zhong et al. (2018)	RL	96.46	39.8	96
Cai et al. (2018)	RL	95.77	23.4	10
Real et al. (2017)	Evolutionary	94.60	5.4	2600
Liu et al. (2018)	Evolutionary	96.25	15.7	300
Thomas Elsken (2018)	Evolutionary	94.80	19.7	1
Zhang et al. (2019)	One-shot	97.16	5.7	0.84
Pham et al. (2018)	One-shot	96.46	4.6	0.5
Bender et al. (2018)	One-shot	96.00	5.0	N/A
Brock et al. (2018)	One-shot	95.97	16.0	3
Liu et al. (2019)	One-shot	97.24	3.3	4
Xie et al. (2019)	One-shot	97.15	2.8	1.5
ILNAS (5.0M)	Gradient-based	**97.42**	**5.0**	**0.8**
ILNAS (2.5M)	Gradient-based	**95.88**	**2.5**	**0.5**
ILNAS (1.0M)	Gradient-based	**94.64**	**1.0**	**0.3**
ILNAS (0.5M)	Gradient-based	**93.75**	**0.5**	**0.2**

Table 1: Comparison of our results with recent works in neural architecture search on the CIFAR-10 dataset.

(a) Results for fixed and random baselines.



(b) Results for different A_{lr}.

Figure 5: Training results compared to a fixed architecture VGG-like network and a NAS algorithm that randomly adds new parameters, and training results with various architecture learning rates.

different search space. In addition to accuracy results, our method is efficient in training time and performs well under weight parameter restrictions.

ImageNet. The ImageNet dataset is far more difficult to classify with its larger input dimensions and number of output classes. We achieve a 76.6% Top-1 / 92.5% Top-5 accuracy with a restriction of 8M parameters and 70.7% Top-1 / 87.9% Top-5 accuracy with a restriction of 2.5M parameters. As Table 2 shows, our results here are similarly competitive with other NAS approaches.

5.2 Architecture Transfer and Multi-Task Learning

We evaluate the potential for transfer learning by adapting a trained network architecture from one dataset to a different classification task. Since our training method changes architecture incrementally at each step, we can continuously evolve the previously trained architecture rather than starting from a random base network.

STL-10. The STL-10 dataset has the same 10 classes as CIFAR-10 but its images have dimension 96×96. We evaluate the network trained on CIFAR-10 with 5M parameters and resize STL-10 images to 32×32 to match input dimensions. Without further training, we achieve 81.8% accuracy. We then train using the 5K provided images and allow the network to expand to achieve 83.4% with 6.0M parameters and 85.6% with 8.0M parameters.

Work	Method Type	Accuracy (Top-1 / 5%)	Params (M)	GPU Days
Zoph et al. (2018)	RL	82.7 / 96.2	88.9	2000
Zhong et al. (2018)	RL	77.4 / 93.5	N/A	96
Cai et al. (2018)	RL	74.6 / 91.9	594	200
Bender et al. (2018)	One-shot	75.2 / N/A	11.9	N/A
Liu et al. (2018)	One-shot	73.3 / 91.3	4.7	4
Zhang et al. (2019)	One-shot	73.0 / 91.3	6.1	0.8
Brock et al. (2018)	One-shot	61.4 / 83.7	16.2	3
Liu et al. (2019)	One-shot	73.3 / 91.3	4.7	4
Xie et al. (2019)	One-shot	72.7 / 90.8	4.3	1.5
ILNAS (8.0M)	Gradient-based	**76.6 / 92.5**	**8.5**	**2**
ILNAS (5.0M)	Gradient-based	**74.1 / 91.4**	**5.0**	**1.5**
ILNAS (2.5M)	Gradient-based	**70.7 / 87.9**	**2.5**	**1**

Table 2: Comparison of results with recent works in NAS on the ImageNet dataset.

Datasets	Shared Params (%)	Total Params	Test Accuracies
(CIFAR, MNIST)	20.0%	5.0M	(95.54%, 99.50%)
(CIFAR, FMNIST)	20.0%	5.0M	(95.16%, 94.23%)
(CIFAR, FMNIST)	33.3%	4.5M	(94.12%, 94.45%)
(ImageNet, CIFAR)	33.3%	7.5M	(71.2% / 87.4%, 92.36%)

Table 3: Training results of multiple datasets on one network with partially shared weight parameters.

MNIST and Fashion-MNIST. Both datasets MNIST and Fashion-MNIST use grayscale input images of dimension 28×28 and 10 output classes. Transferring even our smallest trained CIFAR-10 architecture we observe accuracies $> 94\%$ on Fashion-MNIST and $> 99\%$ on MNIST within 100 epochs of regular training. Doing the reverse by evolving a pre-trained architecture for Fashion-MNIST on CIFAR-10 (with grayscale images), we achieve 90.9% accuracy within 150 epochs, with 2.5M parameters.

Training on two datasets. We simulataneously train two datasets on one network with weights partially shared but outputs separated. The ILNAS algorithm is independently applied to the network architectures dedicated to each task and the shared architecture. Table 3 shows training results of various dataset pairs. For MNIST and FMNIST images, we duplicate inputs for three color channels, and for CIFAR, we upscale input images when the network is shared with ImageNet.

6 Conclusion

We proposed a neural architecture search algorithm that uses local weight gradients to selectively expand the current network's architecture. The main contribution of this work is its different search space consisting of the network's individual connections and nodes that combine to form irregular layers unbounded in both width and depth. Our algorithm trains in a two step cycle where each step isolates either the current weights or architecture and adjusts the other in a similar fashion to the *k*-means algorithm.

Our results show that training by architecture expansion at the level of individual parameters is efficient and achieves results on par with most recent works in NAS. Furthermore, we demonstrate that architectures trained for one dataset can be continuously evolved or restructured to adapt to a different task, and that we can adapt the ILNAS algorithm to train multiple tasks on one network with shared parameters. Since our search space is fundamentally different from that of many related approaches, it is possible to combine it with other methods that search for different aspects of network architecture.

REFERENCES

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 550–559, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/bender18a.html.

Andrew Brock, Theo Lim, J.M. Ritchie, and Nick Weston. SMASH: One-shot model architecture search through hypernetworks. In *6th International Conference on Learning Representations, ICLR 2018*, 2018. URL https://openreview.net/forum?id=rydeCEhs-.

Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 678–687, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/cai18a.html.

Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In Geoffrey Gordon, David Dunson, and Miroslav Dudík (eds.), *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pp. 215–223, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL http://proceedings.mlr.press/v15/coates11a.html.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research 20 (2019) 1-21*, 2018.

Yesmina Jaafra, Jean Luc Laurent, Aline Deruyver, and Mohamed Saber Naceur. A review of meta-reinforcement learning for deep neural networks architecture search, 2018.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Yann LeCun and Corinna Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010. URL http://yann.lecun.com/exdb/mnist/.

Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. doi: 10.1038/nature14539. URL https://doi.org/10.1038/nature14539.

Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv*, 2019.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *6th International Conference on Learning Representations, ICLR 2018*, 2018.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=S1eYHoC5FX.

Renqian Luo, Fei Tian, Tao Qin, En-Hong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in neural information processing systems*, 2018.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pp. 4092–4101, 2018. URL http://proceedings.mlr.press/v80/pham18a.html.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pp. 2902–2911, 2017. URL http://proceedings.mlr.press/v70/real17a.html.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115 (3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pp. 1–37, June 1992. doi: 10.1109/COGANN.1992.273950.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. URL http://nn.cs.utexas.edu/?stanley:ec02.

Frank Hutter Thomas Elsken, Jan Hendrik Metzen. Simple and efficient architecture search for convolutional neural networks. In *6th International Conference on Learning Representations, ICLR 2018*, 2018. URL https://openreview.net/forum?id=SySaJ0xCZ.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms, 2017.

Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: stochastic neural architecture search. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=rylqooRqK7.

Chris Zhang, Mengye Ren, and Raquel Urtasun. Graph hypernetworks for neural architecture search. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=rkgW0oA9FX.

Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017*, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *CVPR*, pp. 8697–8710. IEEE Computer Society, 2018.