EvoGAN: Evolutionary neural architecture search for generative adversarial networks

John Yaras
can.yaras@duke.edu

Baran Yildirim
baran.yildirim@duke.edu

Abstract

Neural architecture search (NAS) is a subfield of automated machine learning (AutoML) that aims to automate the design process of deep neural networks. While there exists a multitude of techniques designed for performing NAS on classification-based networks, comparably few methods exist for automating the search for generative adversarial network (GAN) architectures. In particular, none of the existing approaches use an evolution-based algorithm for designing the two networks involved in the GAN set-up. Our approach, which he call EvoGAN, employs the use of an evolutionary algorithm as a search strategy for designing GAN architectures. Our results show that EvoGAN is able to find an architecture that yields a competitive inception score (IS) on CIFAR-10, while taking much less time to do so compared to other NAS approaches. The implementation and code for EvoGAN is available at `https://github.com/baranyildirim/EvoGAN`.

1 Introduction

In the past decade, deep neural networks have achieved remarkable success on challenging tasks in computer vision, natural language processing, among various other fields. One key aspect of their success involves specific choices in architecture of these networks. However, to solve increasingly difficult tasks, these models have grown exponentially larger over time - modern deep neural networks can have up to billions of parameters [1]. Accordingly, it is becoming difficult for human researchers to hand-craft optimal deep neural networks - neural architecture search (NAS) aims to solve this problem by automating this design process. It has already been shown that NAS applied to classification models can achieve higher accuracy than either random search or manual design [2][3].

On the other hand, there is scarce work in the literature on applying NAS to the design of the generator and discriminator networks involved in the GAN formulation. AutoGAN [3] is a recent approach that is one of the first techniques to apply NAS to GANs. Due to the lack of existing work combining NAS and GANs, we develop an evolutionary algorithm called EvoGAN that searches for an optimal GAN architecture. Evolutionary algorithms are a relatively old approach to optimization problems - they have even been utilized in the realm of neural architecture search, but their application to GAN architecture search is novel.

1.1 Neural Architecture Search (NAS)

The NAS formulation involves 3 entities: the search space, the search strategy, and the performance estimation strategy [2]. Since NAS is a meta-learning problem, these components are reminiscent of a generic optimization problem.

- The search space refers to the set of architectures that our search strategy can explore - it does *not* include the space of trainable parameters of the network. In the context of conventional learning problems, it is analogous to the hypothesis space.

- The search strategy refers to the algorithm which dictates the manner in which the search space is explored. The search strategy is akin to gradient descent for training neural networks. The search strategy is often iterative and utilizes the performance estimation strategy to determine the subsequent elements of the search space to be explored.
- The performance estimation strategy refers to the evaluation method used to predict how well the network will perform after being trained. This is essentially the objective that the search strategy aims to optimize.

1.2 Generative Adversarial Network (GAN)

First introduced by Goodfellow et al. in 2014 [4], generative adversarial networks (GANs) have been very successful in generating synthetic examples of imagery from latent distributions, and have been employed in various other applications, such as text-to-image translation, super resolution, image inpainting, and more. In the Vanilla GAN formulation, we have two networks: a *generator* and a *discriminator*.

- The generator network $G : Z \to X$ takes some latent variable sampled from some latent distribution p_Z over Z as an input and generates a sample in the image space X.
- The discriminator network $D : X \to (0, 1)$ takes a sample image from X and produces some probability.

G, together with the latent distribution p_Z, induces a distribution p_{fake} over X. The goal of G is to match the distribution of synthetically generated images p_{fake} to the distribution of real images p_{real}. On the other hand, the goal of D is to distinguish images drawn from p_{fake} from those drawn from p_{real}. These two networks are trained jointly in a two-player minimax game with objective

$$\min_G \max_D \mathbb{E}_{x \sim p_{\text{real}}} \left[\log D(x)\right] + \mathbb{E}_{z \sim p_Z} \left[\log \left\{1 - D(G(z))\right\}\right]. \tag{1}$$

Training D and G using the objective in (1) is highly susceptible to convergence issues, since gradient-based optimization methods are not guaranteed to find the Nash equilibrium of a two-player game. Furthermore, vanishing gradients can occur when the discriminator is performing substantially better than the generator, in which case the generator effectively cannot learn. Mode collapse is another common issue with Vanilla GAN, in which the generator maps large regions of the latent space Z to a small region in image space X, i.e. the synthetic images produced by the generator are all very similar. Issues such as vanishing gradients and mode collapse are alleviated by other approaches, such as Wasserstein GAN [5].

2 Related Work

2.1 Search Space

Much of the earlier literature in NAS considers the search space of single branch networks, which consist of a multiple layers composed sequentially. Architectures in this search space are typically characterized by the number of layers in the network, the operation performed by each layer, e.g. basic convolution, dilated convolution [6], or depthwise separable convolution [7], or the specific parameters of the operation type, such as the number of convolutions or the width of the convolutional kernel [8].

More recent work considers multiple branch networks [9], in which several operations may be performed in parallel - single branch networks are contained in this new search space. While this generalization results in a considerably larger search space, it allows for the possibility of finding residual network architectures, which have been shown to improve convolutional neural networks considerably.

Another approach to modeling the search space is considering the space of *cell* architectures, which are assembled in a predetermined manner to create the final network [10]. This idea is motivated by the design pattern of composing abstract building blocks when hand-crafting networks. One advantage of this approach includes a simpler search space, since a single cell has fewer layers than an entire network. Furthermore, the discovered cell architecture can easily be transferred to

other computer vision tasks. However, this approach includes human bias in the macro-architecture - several approaches jointly optimize the micro-architecture (cell design) and the macro-architecture (cell placements) [11].

2.2 Search Strategy

The search strategy can be naturally formulated as a reinforcement learning problem [8], where inferring the network architecture is the action of the agent, and the reward is derived from the performance estimation strategy (e.g. performance of the architecture on some validation set). The policy of the agent can be modeled in several ways, such as using a recurrent neural network policy [12] to sample a string encoding the architecture, or using Q-learning [8] to sequentially sample each layer's operation type and shape. All these approaches can be summarized as various implementations of a multi-armed bandit problem.

A relatively old search strategy involves biologically inspired evolutionary algorithms for navigating the search space [13]. These algorithms generally consist of tracking a population of samples (i.e. architectures) from the search space, some subset of which are used to breed new offsprings in the next generation. Generally, the parents are chosen using some measure of fitness, which is again related to the performance estimation strategy, and usually consists of evaluating performance on some validation set. Approaches to choosing parents for breeding include tournament selection [14] or solving a multi-objective optimization problem [15]. Studies in the literature have shown that reinforcement learning and evolutionary algorithm perform similarly on most NAS problems, with evolutionary algorithms generally having faster run-time and finding smaller models [16].

Several less popular approaches include, for example, Bayesian optimization [17], which excels at optimizing objectives that are expensive to compute, and have therefore been applied to NAS problems. Other sequential model-based optimization algorithms include tree Parzen estimators, which have conventionally been used for hyperparameter selection [18]. More recent approaches convert the discrete search space into a continuous one and perform gradient-based optimization [19].

2.3 Performance Estimation Strategy

The simplest approach to estimating the performance of a particular architecture involves full training of the model, followed by evaluation on some validation set. However, this strategy is quite expensive - therefore less costly approaches have been developed. Most commonly, models are trained for fewer epochs [20], smaller datasets [21], smaller image sizes [22], etc. This approach drastically reduces training time, but inferring the performance of a fully-trained network from a smaller subproblem may be risky. In light of this, several approaches attempt to correctly extrapolate the training curve from the first few epochs [23]. For evolution-based algorithms, offspring can be initialized with weights inherited from parent models, which is essentially performing transfer learning across slightly different models [14]. Finally, one-shot architecture search [9] considers all architectures of the search space as subgraphs of a single fully-connected graph - then weights are inherited between architectures that are adjacent in the supergraph. This implies that only a single model in the search space needs to be fully trained, since these weights can be propagated throughout the graph to infer parameters for different architectures.

2.4 AutoGAN

AutoGAN [3] is a relatively recent approach that introduces NAS methodologies to GAN architecture design. First of all, AutoGAN only searches for the architecture of the generator network. The search space consists of the set of possible cell architectures, which vary in the use of skip connections/shortcuts, convolution type, normalization type, and upsampling type. These cells are then composed sequentially to build the final inferred architecture. In their experiments, the cell count is fixed, so there are 3 cells in total. For the search strategy, AutoGAN utilizes an RNN controller, sequentially predicts each cell architecture. This controller is trained via the REINFORCE algorithm that utilizes inception score as a reward. For the performance estimation strategy, each inferred model is trained using a dynamic-resetting approach, in which training is stopped when the generator is no

longer training due to convergence, or issues such as mode collapse. This is detected using a moving window that tracks the training statistics of the generator.

3 EvoGAN

In EvoGAN, we use the same cell search space used in AutoGAN. More specifically, the search space is specified at the cell-level, where the s-th cell is defined by $s + 4$ parameters:

- s values indicating incoming skip connections from previous cells
- basic convolution block type (0: pre-activation, 1: post-activation)
- normalization type (0: no normalization, 1: instance normalization, 2: batch normalization)
- upsampling type (0: bilinear, 1: nearest neighbor, 2: deconvolution)
- presence of an in-cell shortcut

We fix the number of cells to 3, so the entire architecture is specified by $4 + 5 + 6 = 15$ parameters, which come from a variable number of parameter options. By enumerating each architecture choice, we can encode each architecture as a string of length 15 - we call this representation the DNA of the architecture.

EvoGAN uses an evolutionary algorithm as a search strategy. First, we randomly generate an initial population of DNA from uniform distributions over each DNA parameter. Each architecture in the population is then assigned fitness via its inception score, which are then used to construct new distributions over the DNA parameters. The new distributions are constructed so that parameters belonging to DNA with high fitness are assigned a higher probability. These distributions are then used to generate a new population in the evolution step. To encourage early exploration of the search space, there is a small non-zero probability that DNA parameters can mutate into different parameters after each evolution - this mutation probability decays by some multiplicative factor each generation.

Our performance estimation strategy uses inception score to evaluate each architecture. To reduce the cost of scoring each DNA, we train each architecture for a single epoch before scoring. We use the same discriminator that is used in AutoGAN, namely 4 convolutional blocks consisting of 2 convolutional layers with ReLU activations inbetween. The objective used for training the GAN is the hinge loss also used in AutoGAN. More specifically, the loss for the discriminator is given by

$$\mathcal{L}_D = \mathbb{E}_{x\sim p_{real}} \min(0, -1 + D(x)) + \mathbb{E}_{z\sim pZ} \min(0, -1 - D(G(z))) \tag{2}$$

and the loss for the generator is given by

$$\mathcal{L}_G = \mathbb{E}_{z\sim pZ} \min(0, D(G(z))). \tag{3}$$

The full EvoGAN algorithm can be found in the appendix at the end.

4 Results

In our experiments, we ran EvoGAN using the following parameters:

$$T = 10 \quad N = 5 \quad p_{mut} = 0.1 \quad \alpha = 1/3.$$

The generator discovered by EvoGAN is shown in Figure 3, and a comparison between the EvoGAN architecture and other GANs can be found in Figure 1.

It took a little less than 2 hours[1] for the population to converge on a single architecture. This architecture was then trained for 640 epochs, and achieved an inception score of 7.96 and a FID score of 15.91 on CIFAR-10. As shown in Figure 1, this architecture is competitive at the task of unconditional image generation even when compared to handcrafted GANs.

The evolutionary algorithm approach is clearly extremely efficient at finding optimal architectures, as the convergence time is quiet short. In comparison, AutoGAN was trained for 42 hours[2] [3], and achieved an inception score of 8.55 and a FID score of 12.42 on CIFAR-10.

[1]Experiments were conducted on a machine with a single NVIDIA GTX 1070 graphics card

[2]No details about the training system were specified, but it can be inferred from the RNN architecture that the computational requirements for AutoGAN are high.

Method	IS	FID
DCGAN [24]	6.64	-
Improved GAN [25]	6.86	-
LRGAN [26]	7.17	-
DFM [27]	7.72	-
ProbGAN [28]	7.75	24.60
WGAN-GP, ResNET [29]	7.86	-
Splitting GAN [30]	7.90	-
SN-GAN [31]	8.22	21.7
MGAN [32]	8.33	26.7
Dist-GAN [33]	-	17.61
Progressive GAN [34]	8.80	-
Improving MMD GAN [35]	8.29	16.21
AutoGAN [3]	8.55	12.42
EvoGAN	**7.96**	**15.91**

Figure 1: Inception score and FID score for unconditional image generation task on CIFAR-10

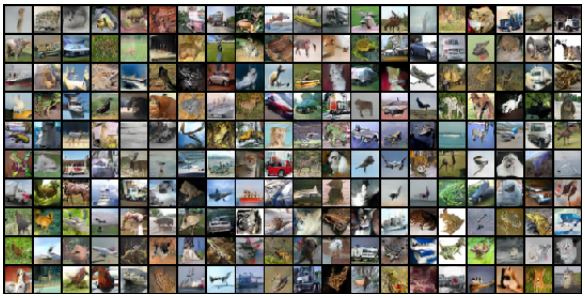

Figure 2: Image samples generated by the best architecture after training

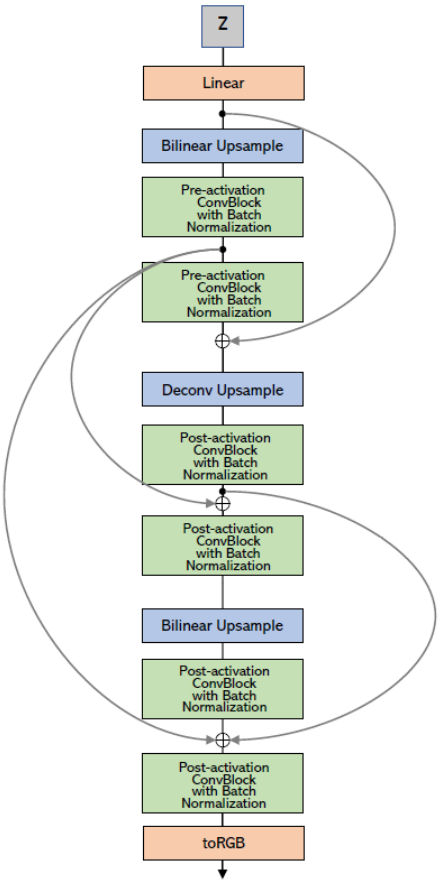

Figure 3: The generator architecture discovered by EvoGAN on CIFAR-10

The generator architecture found by EvoGAN on CIFAR-10 has several interesting properties:

- The architecture has batch normalization in all convolution blocks.
- Skip connections exist in every position possible.
- There are pre-activation convolution blocks in the first cell and post-activation convolution blocks in the second and third cells.
- An in-cell shortcut is present only in the first cell.
- A bilinear upsample is present in the first and third cell, while a deconvolution upsample is present in the second cell.

5 Discussion

EvoGAN offers an efficient neural architecture search approach for GANs. EvoGAN is able to identify an effective architecture on the CIFAR-10 dataset, while taking much less time than the alternative neural architecture search approach in AutoGAN [3]. This outcome is corroborated by the studies in [16], which discover that evolutionary algorithms often have faster runtimes than similar performing reinforcement learning approaches. Furthermore, the EvoGAN architecture was found to be competitive at the task of unconditional image generation even when compared to other handcrafted GANs.

Another advantage of our method is that it is a highly scalable approach that can parallelize well over a large amount of computation resources. In theory, we could simultaneously train all architectures in the population of each generation. This would also allow us to train each architecture for more than 1 epoch to yield a less coarse fitness score.

Given more time and resources, we would've liked to explore searching for the generator and discriminator architectures simultaneously. To do so, we could first consider encoding some predetermined discriminator search space into DNA string representations, as we did with the generator architecture search space - then we can concatenate the DNA string of each discriminator with the DNA string of each generator to form a new search space that is formally the product space of the discriminator and generator search spaces. In this way, we only need to extend our DNA strings, but otherwise EvoGAN can be naturally extended to optimizing both architectures jointly.

Furthermore, it is important to note that the experiments were carried out on the CIFAR-10 dataset, which is an academic dataset with small images. This allows a performance comparison to be easily drawn between architectures found by EvoGAN and various other architectures. However, performance should be measured on various other datasets to get a complete picture of EvoGAN's effectiveness.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[2] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. 2018.

[3] Xinyu Gong, Shiyu Chang, Yifan Jiang, and Zhangyang Wang. Autogan: Neural architecture search for generative adversarial networks. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2019.

[4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[5] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017.

[6] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions, 2015.

[7] François Chollet. Xception: Deep learning with depthwise separable convolutions, 2016.

[8] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning, 2016.

[9] Andrew Brock, Theodore Lim, J. M. Ritchie, and Nick Weston. Smash: One-shot model architecture search through hypernetworks, 2017.

[10] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation, 2017.

[11] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search, 2017.

[12] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2016.

[13] G. Miller, P. Todd, and S. Hegde. Designing neural networks using genetic algorithms. In *ICGA*, 1989.

[14] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers, 2017.

[15] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution, 2018.

[16] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search, 2018.

[17] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric Xing. Neural architecture search with bayesian optimisation and optimal transport. 2018.

[18] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123, 2013.

[19] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search, 2018.

[20] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition, 2017.

[21] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. volume 54 of *Proceedings of Machine Learning Research*, pages 528–536, Fort Lauderdale, FL, USA, 20–22 Apr 2017. PMLR.

[22] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the cifar datasets, 2017.

[23] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw bayesian optimization, 2014.

[24] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2015.

[25] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016.

[26] Jianwei Yang, Anitha Kannan, Dhruv Batra, and Devi Parikh. Lr-gan: Layered recursive generative adversarial networks for image generation, 2017.

[27] David Warde-Farley and Yoshua Bengio. Improving generative adversarial networks with denoising feature matching. In *ICLR*, 2017.

[28] Hao He, Hao Wang, Guang-He Lee, and Yonglong Tian. Bayesian modelling and monte carlo inference for GAN. In *International Conference on Learning Representations*, 2019.

[29] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans, 2017.

[30] Guillermo L. Grinblat, Lucas C. Uzal, and Pablo M. Granitto. Class-splitting generative adversarial networks, 2017.

[31] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks, 2018.

[32] Quan Hoang, Tu Dinh Nguyen, Trung Le, and Dinh Phung. MGAN: Training generative adversarial nets with multiple generators. In *International Conference on Learning Representations*, 2018.

[33] Ngoc-Trung Tran, Anh Bui, and Ngai-Man Cheung. Dist-gan: An improved gan using distance constraints. 03 2018.

[34] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation, 2017.

[35] Wei Wang, Yuan Sun, and Saman Halgamuge. Improving mmd-gan training with repulsive loss function, 2018.

Appendix

Algorithm 1 EvoGAN

Input: Max number of evolutions T, DNA samples per generation N, DNA parameter options P, initial mutation probability p_{mut}, mutation probability decay rate α

Output: Top K architecture DNAs

for $i \leftarrow 1$ to N **do**
  for $j \leftarrow 1$ to $|P|$ **do**
    $d_{ij}^{(0)} \leftarrow \text{sample}(\text{Unif}(P_j))$ ▷ Initialize random population uniformly from P
  end for
end for
for $t \leftarrow 1$ to T **do**
  for $j \leftarrow 1$ to $|P|$ **do**
    for $k \leftarrow 1$ to $|P_j|$ **do**
      $z_{jk}^{(t)} \leftarrow 0$ ▷ Initialize reward of parameter j due to parameter option k
    end for
  end for
  for $i \leftarrow 1$ to N **do**
    $s_i^{(t)} \leftarrow \text{fitness}\left(d_i^{(t-1)}\right)$ ▷ Get fitness (IS) of architecture with DNA $d_i^{(t-1)}$
    for $j \leftarrow 1$ to $|P|$ **do**
      $z_{jd_{ij}^{(t-1)}}^{(t)} \leftarrow z_{jd_{ij}^{(t-1)}}^{(t)} + \exp\left(s_i^{(t)}\right)$ ▷ Reward parameter options based on DNA fitness
    end for
  end for
  for $j \leftarrow 1$ to $|P|$ **do**
    for $k \leftarrow 1$ to $|P_j|$ **do**
      $p_{jk}^{(t)} \leftarrow z_{jk}^{(t)} / \sum_{l=1}^{|P_j|} z_{jl}^{(t)}$ ▷ Normalize the rewards of each parameter into probabilities
    end for
  end for
  for $i \leftarrow 1$ to N **do**
    for $j \leftarrow 1$ to $|P|$ **do**
      $d_{ij}^{(t)} \leftarrow \text{sample}(\text{Cat}(P_j, p_j^{(t)}))$ ▷ Generate new population from distribution $p_j^{(t)}$
      mutate $d_{ij}^{(t)}$ with probability $\alpha^{t-1} p_{mut}$
    end for
  end for
end for