GENERATIVE TEACHING NETWORKS: ACCELERATING NEURAL ARCHITECTURE SEARCH BY LEARNING TO GENERATE SYNTHETIC TRAINING DATA

Anonymous authors
Paper under double-blind review

ABSTRACT

This paper investigates the intriguing question of whether we can create learning algorithms that automatically generate training data, learning environments, and curricula in order to help AI agents rapidly learn. We show that such algorithms are possible via Generative Teaching Networks (GTNs), a general approach that is applicable to supervised, unsupervised, and reinforcement learning. GTNs are deep neural networks that generate data and/or training environments that a learner (e.g. a freshly initialized neural network) trains on before being tested on a target task. We then differentiate *through the entire learning process* via meta-gradients to update the GTN parameters to improve performance on the target task. GTNs have the beneficial property that they can theoretically generate any type of data or training environment, making their potential impact large. This paper introduces GTNs, discusses their potential, and showcases that they can substantially accelerate learning. We also demonstrate a practical and exciting application of GTNs: accelerating the evaluation of candidate architectures for neural architecture search (NAS), which is rate-limited by such evaluations, enabling massive speed-ups in NAS. GTN-NAS improves the NAS state of the art, finding higher performing architectures when controlling for the search proposal mechanism. GTN-NAS also is competitive with the overall state of the art approaches, which achieve top performance while using orders of magnitude less computation than typical NAS methods. Overall, GTNs represent a first step toward the ambitious goal of algorithms that generate their own training data and, in doing so, open a variety of interesting new research questions and directions.

1 INTRODUCTION AND RELATED WORK

Access to a large amount of training data is now common in many machine learning (ML) problems. However, to effectively train neural networks (NNs) does not require using *all available* data. For example, recent work in curriculum learning (Graves et al., 2017), active learning (Konyushkova et al., 2017; Settles, 2010) and core-set selection (Sener & Savarese, 2018; Tsang et al., 2005) demonstrates that a surrogate dataset can be created by intelligently sampling a subset of training data, and that such surrogates enable competitive test performance with less training effort. Being able to more rapidly determine the performance of an architecture in this way could particularly benefit architecture search, where training thousands or millions of candidate NN architectures on full datasets can become prohibitively expensive. From this lens, related work in learning-to-teach has shown promise. For example, the learning to teach (L2T) (Fan et al., 2018) method accelerates learning for a NN learner (hereafter, just *learner*) through reinforcement learning, by learning how to subsample mini-batches of data.

A key insight in this paper is that the surrogate data need not be drawn from the original data distribution (i.e. they may not need to resemble the original data). For example, humans can learn new skills from reading a book or can prepare for a team game like soccer by practicing skills, such as passing, dribbling, juggling, and shooting. This paper investigates the question of whether we can train a data-generating network that can produce *synthetic* data that effectively and efficiently teaches a target task to a learner. Related to the idea of generating data, Generative Adversarial Networks (GANs) can produce impressive high-resolution images (Goodfellow et al., 2014; Brock et al., 2018), but they are incentivized to mimic real data (Goodfellow et al., 2014), instead of being optimized to teach learners *more* efficiently than real data.

Another approach for creating surrogate training data is to treat the training data itself as a hyper-parameter of the training process and learn it directly. Such learning can be done through meta-gradients (also called hyper-gradients), i.e. differentiating through the training process to optimize a meta-objective. This approach was described in Maclaurin et al. (2015), where 10 synthetic training images were learned using meta-gradients such that when a network is trained on these images, the network's performance on the MNIST validation dataset is maximized. In recent work concurrent with our own, Wang et al. (2019b) scaled this idea to learn 100 synthetic training examples. While the 100 synthetic examples were more effective for training than 100 original (real) MNIST training examples, we show that it is difficult to scale this approach much further without the regularity across samples provided by a generative architecture (Figure 3b, green line).

Being able to very quickly train learners is particularly valuable for neural architecture search (NAS), which is exciting for its potential to automatically discover high-performing architectures, which otherwise must be undertaken through time-consuming manual experimentation for new domains. Many advances in NAS involve accelerating the evaluation of candidate architectures by training a predictor of how well a trained learner would perform, by extrapolating from previously trained architectures (Luo et al., 2018; Liu et al., 2018a; Baker et al., 2017). This approach is still expensive because it requires many architectures to be trained and evaluated to train the predictor. Other approaches accelerate training by sharing training across architectures, either through shared weights (e.g. as in ENAS; Pham et al. (2018)), or Graph HyperNetworks (Zhang et al., 2018).

We propose a scalable, novel, meta-learning approach for creating synthetic data called Generative Teaching Networks (GTNs). GTN training has two nested training loops: an inner loop to train a learner network, and an outer-loop to train a generator network that produces synthetic training data for the learner network. Experiments presented in Section 3 demonstrate that the GTN approach produces synthetic data that enables much faster learning, speeding up the training of a NN by a factor of 9. Importantly, the synthetic data in GTNs is not only agnostic to the weight initialization of the learner network (as in Wang et al. (2019b)), but is also agnostic to the learner's *architecture*. As a result, GTNs are a viable method for accelerating evaluation of candidate architectures in NAS. Indeed, controlling for search algorithm, GTN-NAS improves the NAS state of the art by finding higher-performing architectures than comparable methods like weight sharing (Pham et al., 2018) and Graph HyperNetworks (Zhang et al., 2018); it also is competitive with methods using more sophisticated search algorithms and orders of magnitude more computation. It could also be combined with those methods to provide further gains.

One promising aspect of GTNs is that they make very few assumptions about the learner. In contrast, NAS techniques based on shared training are viable only if the parameterizations of the learners are similar. For example, it is unclear how weight-sharing or HyperNetworks could be applied to architectural search spaces wherein layers could be either convolutional or fully-connected, as there is no obvious way for weights learned for one layer type to inform those of the other. In contrast, GTNs are able to create training data that can generalize between such diverse types of architectures.

GTNs also open up interesting new research questions and applications to be explored by future work. Because they can rapidly train new architectures, GTNs could be used to create NNs *on-demand* that meet specific design constraints (e.g. a given balance of performance, speed, and energy usage) and/or have a specific subset of skills (e.g. perhaps one needs to rapidly create a compact network capable of three particular skills). Because GTNs can generate virtually any learning environment, they also one day could be a key to creating AI-generating algorithms, which seek to bootstrap themselves from simple initial conditions to powerful forms of AI by creating an open-ended stream of challenges (learning opportunities) while learning to solve them (Clune, 2019).

2 METHODS

The main idea in GTNs is to train a data-generating network such that a learner network trained on data it produces achieves high performance in a target task. Unlike a GAN, here the two players (networks) cooperate (rather than compete) because their interests are aligned towards having the learner perform well on the target task when trained on data produced by the GTN. The generator and the learner networks are trained with meta-learning via nested optimization that consists of inner and outer training loops (Figure 1). In the inner-loop, the generator $G(z, y)$ takes Gaussian noise (z) and a label (y) as input and outputs synthetic data (x). Optionally, the generator could take only noise as input and produce both data and labels as output (Appendix F). The learner is then trained on this synthetic data for a fixed number of inner-loop training steps with any optimizer, such as SGD or Adam (Kingma & Ba, 2014): we use SGD with momentum in this paper. Equation 1 defines the inner-loop SGD with momentum update for the learner parameters θ_t.

$$\theta_{t+1} = \theta_t - \alpha \sum_{0 \le t' \le t} \beta^{t-t'} \nabla \ell_{\text{inner}}(G(\mathbf{z}_{t'}, \mathbf{y}_{t'}), \mathbf{y}_{t'}, \theta_{t'}), \tag{1}$$

where α and β are the learning rate and momentum hyperparameters, respectively. We sample $\mathbf{z}_t$ from a unit-variance Gaussian and $\mathbf{y}_t$ uniformly from all available class labels. Note that both $\mathbf{z}_t$ and $\mathbf{y}_t$ are batches of samples. We can also learn a curriculum directly by additionally optimizing $\mathbf{z}_t$ directly (instead of sampling it randomly) and keeping $\mathbf{y}_t$ fixed throughout all of training.

The inner-loop loss function ℓ_{inner} can be cross-entropy for classification problems or mean squared error for regression problems. Note that the inner-loop objective does not depend on the outer-loop objective and could even be parameterized and learned through meta-gradients with the rest of the system (Houthooft et al., 2018). In the outer-loop, the learner θ_T (i.e. the learner parameters trained on *synthetic* data after the T inner-loop steps) is evaluated on the real *training* data, which is used to compute the outer-loop loss (aka meta-training loss). The gradient of the meta-training loss with respect to the generator is computed by backpropagating through the entire inner-loop learning process. While computing the gradients for the generator we also compute the gradients of hyper-parameters of the inner-loop SGD update rule (its learning rate and momentum), which are updated after each outer-loop at no additional cost. To reduce memory requirements, we leverage gradient-checkpointing (Griewank & Walther, 2000) when computing meta-gradients. The computation and memory complexity of our approach can be found in Appendix D.

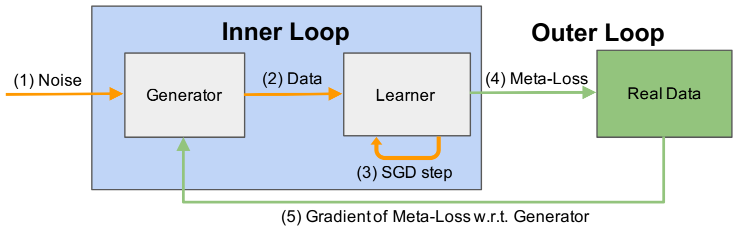


Figure 1: Generative Teaching Network (GTN) Method. The numbers in the figure reflect the order in which a GTN is executed. Noise is fed as an input to the Generator (1), which uses it to generate new data (2). The learner is trained (e.g. using SGD or Adam) to perform well on the generated data (3). The trained learner is then evaluated on the real training data in the outer-loop to compute the outer-loop meta-loss (4). The gradients of the generator parameters are computed w.r.t. to the meta-loss to update the generator (5).

A key motivation for this work is to generate synthetic data that is learner agnostic, i.e. that generalizes across different potential learner architectures and initializations. To achieve this objective, at the beginning of each new outer-loop training, we choose a new learner architecture according to a predefined set (Section 3) and randomly initialize it (details in Appendix A).

Meta-learning with Weight Normalization. Optimization through meta-gradients is often unstable (Maclaurin et al., 2015). We observed that this instability greatly complicates training because of its hyperparameter sensitivity, and training quickly diverges if they are not well-set. Combining the gradients from Evolution Strategies (Salimans et al., 2017) and backpropagation using inverse variance weighting (Fleiss, 1993; Metz et al., 2019) improved stability in our experiments, but optimization still consistently diverged whenever we increased the number of inner-loop optimization steps. To mitigate this issue, we introduce applying weight normalization (Salimans & Kingma, 2016) to stabilize meta-gradient training by normalizing the generator and learner weights. Instead of updating the weights (W) directly, we parameterize them as $W = g \cdot V / \|V\|$ and instead update the scalar g and vector V. Weight normalization eliminates the need for (and cost of) calculating ES gradients and combining them with backprop gradients, simplifying and speeding up the algorithm.

We hypothesize that weight normalization will help stabilize meta-gradient training more broadly. The idea is that applying weight normalization to meta-learning techniques is analogous to batch normalization for deep networks (Ioffe & Szegedy, 2015). Batch normalization normalizes the forward propagation of activations in a long sequence of parameterized operations (a deep NN). In meta-gradient training both the activations and weights result from a long sequence of parameterized operations and thus both should be normalized. Results in section 3.1 support this hypothesis.

Learning a Curriculum with Generative Teaching Networks. Previous work has shown that a learned curriculum can be more effective than training from uniformly sampled data (Graves et al., 2017). A curriculum is usually encoded with indexes to samples from a given dataset, rendering it non-differentiable and thereby complicating the curriculum's optimization. With GTNs however, a curriculum can be encoded as a series of input vectors to the generator (i.e. instead of sampling the $\mathbf{z}_t$ inputs to the generator from a Gaussian distribution, a sequence of $\mathbf{z}_t$ inputs can be learned). A curriculum can thus be learned by differentiating through the generator to optimize this sequence (in addition to the generator's parameters). Experiments confirm that GTNs more effectively teach learners when optimizing such a curriculum (Section 3.2).

Accelerating NAS with Generative Teaching Networks. Since GTNs can accelerate learner training, we propose harnessing GTNs to accelerate NAS. Rather than evaluating each architecture in a target task with a standard training procedure, we propose evaluating architectures with a meta-optimized training process (that generates synthetic data in addition to optimizing inner-loop hyperparameters). We show that doing so significantly reduces the cost of running NAS (Section 3.4).

The goal of these experiments is to find a high-performing CNN architecture for the CIFAR10 image-classification task (Krizhevsky et al., 2009) with limited compute costs. We use the same architecture search-space, training procedure, hyperparameters, and code from Neural Architecture Optimization (Luo et al., 2018), a state-of-the-art NAS method. The search space consists of the topology of two cells: a reduction cell and a convolutional cell. Multiple copies of such cells are stacked according to a predefined blueprint to form a full CNN architecture (see Luo et al. (2018) for more details). The blueprint has two hyperparameters N and F that control how many times the convolutional cell is repeated (depth) and the width of each layer, respectively. Each cell contains $B = 5$ nodes. For each node within a cell, the search algorithm has to choose two inputs as well as two operations to apply to those inputs. The inputs to a node can be previous nodes or the outputs of the last two layers. There are 11 operations to choose from (Appendix C).

Following Luo et al. (2018), we report the performance of our best cell instantiated with $N = 6, F = 36$ after the resulting architecture is trained for a significant amount of time (600 epochs). Since evaluating each architecture in those settings (named *final evaluation* from now on) is time consuming, Luo et al. (2018) uses a surrogate evaluation (named *search evaluation*) to estimate the performance of a given cell wherein a smaller version of the architecture ($N = 3, F = 32$) is trained for less epochs (100) on real data. We further reduce the evaluation time of each cell by replacing the training data in the search evaluation with GTN synthetic data, thus reducing the training time per evaluation by 300x (which we call *GTN evaluation*). While we were able to train GTNs directly on the complex architectures from the NAS search space, training was prohibitively slow. Instead, for these experiments, we optimize our GTN ahead of time using proxy learners described in Appendix B, which are smaller fully-convolutional networks (this meta-training took 8h on one p6000 GPU). Interestingly, although we never train our GTN on any NAS architectures, because of generalization, synthetic data from GTNs were still effective for training them.

3 Results

We first demonstrate that weight normalization significantly improves the stability of meta-learning, an independent contribution of this paper (Section 3.1). We then show that training with synthetic data is more effective when learning such data jointly with a curriculum that orders its presentation to the learner (Section 3.2). We next show that GTNs can generate a synthetic training set that is more effective than real training data in two supervised learning domains (MNIST and CIFAR10) and in a reinforcement learning domain (cart-pole, Appendix H). We then apply GTN-synthetic training data for neural architecture search to find high performing architectures for CIFAR10 with limited compute, outperforming comperable methods like weight sharing (Pham et al., 2018) and Graph HyperNetworks (Zhang et al., 2018) (Section 3.4).

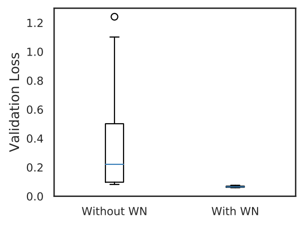
We uniformly split the usual MNIST *training* set into training (50k) and validation sets (10k). The training set was used for inner-loop training (for the baseline) and to compute meta-gradients for all the treatments. We used the validation set for hyperparameter tuning and report performance on the usual MNIST test set (10k images). We followed the same procedure for CIFAR10, resulting in training, validation, and test sets with 45k, 5k, and 10k examples, respectively. Unless otherwise specified, we ran each experiment 5 times and plot the mean and its 95% confidence intervals from (n=1,000) bootstrapping. Appendix A describes the hyperparameters and architectures used.

3.1 Improving Stability with Weight Normalization

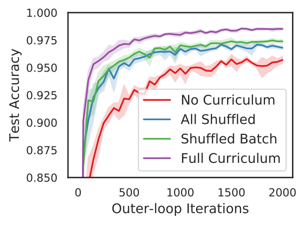
To demonstrate the effectiveness of weight normalization for stabilizing and robustifying meta-optimization, we compare the results of running hyperparameter optimization for GTNs with and without weight normalization on MNIST. Figure 2a shows the distribution of the final performance obtained for 20 runs *during* hyperparameter tuning, which reflects how sensitive the algorithms are to hyperparameter settings. Overall, weight normalization substantially improved robustness to hyperparameters and final learner performance, supporting the initial hypothesis.

3.2 Improving GTNs with a Curriculum

In this section, we experimentally evaluate four different variants of GTNs, each with increasing control over the ordering of the input samples. The first variant (called *GTN - No Curriculum*), trains a generator to output synthetic training data by sampling the noise vector $\mathbf{z}_t$ from a Gaussian distribution at each iteration. In the next three GTN variants, the generator is provided with a fixed set of input samples (instead of a noise vector). These input samples are learned along with the generator parameters during GTN training. For instance, the second GTN variant (called *GTN - All Shuffled*) learns a fixed set of 4,096 input samples that are presented in a random order without replacement. The third variant (called *GTN - Shuffled Batch*) learns 32 batches of 128 samples each, but the order in which the batches are presented is randomized (without replacement). Finally, the fourth variant (called *GTN - Full Curriculum*) learns the exact order in which 32 batches of 128 samples each are presented to the generator. We plot the test accuracy of a learner (with random initial weights and architecture) as a function of outer-loop iterations for all four variants in Figure 2b. Although *GTNs - No curriculum* can seemingly generate endless data (see Appendix G), it performs worse than the other three variants with a fixed set of generator inputs. Overall, training the GTN with exact ordering of input samples (*GTN - Full Curriculum*) outperforms all other variants.



(a) GTN stability with WN



(b) GTN curricula comparison

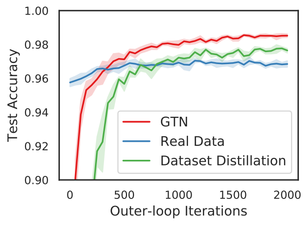
Figure 2: Both a learned curriculum and weight normalization substantially improve GTN performance. (a) Weight normalization improves meta-gradient training of GTNs, and makes the method much more robust to different hyperparameter settings. Each boxplot reports the final loss of 20 runs obtained *during* hyperparameter optimization with Bayesian Optimization (lower is better). (b) shows a comparison between GTNs with different types of curricula. The GTN method with the most control over how samples are presented performs the best.

While curriculum learning usually refers to training on easy tasks first and increasing their difficulty over time, our curriculum goes beyond presenting tasks in a certain order. Specifically, *GTN - Full Curriculum* learns both the order in which to present samples and the specific group of samples to present at the same time. The ability to learn a full curriculum improves GTN performance. For that reason, we adopt that approach for all GTN experiments.

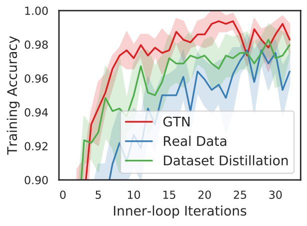
3.3 GTNs for Supervised Learning

To explore whether GTNs can generate useful training data, we compare the performance of 3 meta-learning treatments for MNIST classification. 1) *Real Data* - Training learners with random mini-batches of real data, as is ubiquitous in SGD. 2) *Dataset Distillation* - Training learners with synthetic data, where training examples are directly encoded as tensors optimized by the meta-objective, as in Wang et al. (2019b). 3) *GTN* - Our method where the training data presented to the learner is generated by a neural network. Note that all three methods meta-optimize the inner-loop hyperparameters (i.e. the learning rate and momentum of SGD) as part of the meta-optimization.

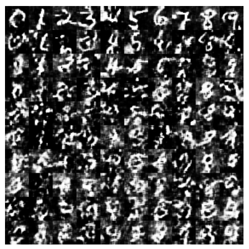
Figure 3a shows that the GTN treatment significantly outperforms the other ones ($p < 0.01$) and trains a learner to be much more accurate when controlling for computation. Specifically, for each treatment the figure shows the test performance of a learner following 32 inner-loop training steps with a batch size of 128. Figure 3b shows the performance of a learner from each treatment after 2000 total outer-loop iterations ($\sim$1 hour using a p6000 GPU). For reference, Dataset Distillation (Wang et al., 2019b) reported 79.5% accuracy for a randomly initialized network (using 100 synthetic images, while we use 4,096) and L2T (Fan et al., 2018) reported needing 300x more training iterations to achieve $> 98\%$ accuracy on MNIST. We show examples of the synthetic images generated by GTN in Figure 3c. Surprisingly, although recognizable as digits and effective for training, GTN-generated images were not visually realistic.



(a) Meta-training curves



(b) Training curves



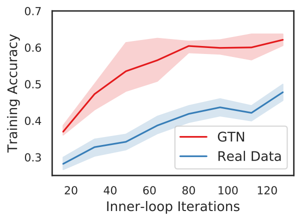
(c) GTN-generated samples

Figure 3: Teaching MNIST with GTN-generated images. (a) shows MNIST test set performance across outer-loop iterations for different sources of inner-loop training data. The inner-loop consists of 32 SGD steps and the outer-loop optimizes MNIST validation performance. Our method (GTN) outperforms the two controls (dataset distillation and samples from real data). (b) shows, given final meta-training iteration, how iterations of inner-loop training compare between training data sources (measured by MNIST training set accuracy). (c) shows 100 random samples from the trained GTN. Samples are usually recognizable as digits, but are not realistic. Each column contains samples from a different digit class, and each row is taken from different inner-loop iterations (evenly spaced from the 32 total iterations, with early iterations at the top).

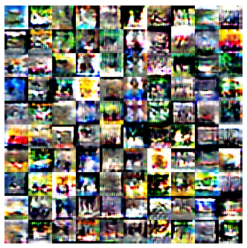
3.4 Architecture Search with GTNs

We next test the benefits of GTN for NAS (GTN-NAS) in CIFAR10, a domain where NAS has previously shown significant improvements over the best architectures produced by armies of human scientists. Figure 4a shows the training performance of a learner trained with either GTN-synthetic data or real data (CIFAR10) over many inner-loop training iterations. After 8h of meta-training, training with GTN-generated data was significantly faster than with real data, as in MNIST.

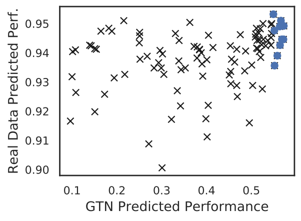
To explore the potential for GTN-NAS to accelerate CIFAR10 architecture search, we investigated the rank correlation (across architectures sampled from the NAS search space) between accelerated GTN-trained network performance (*GTN evaluation*) and the usual more expensive performance metric used during NAS (*search evaluation*). A correlation plot is shown in Figure 4c; note that a strong correlation implies we can train architectures using GTN evaluation as an inexpensive surrogate. We find that GTN evaluation enables predicting the performance of an architecture efficiently. The rank-correlation between 128 *steps* of training with GTN-synthetic data vs. 100 *epochs* of real data is 0.3606. The correlation improves to 0.5582 when considering the top 50% of architectures recommended by GTN evaluation scores, which is important because those are the ones that search would select. This improved correlation is slightly stronger than that from *3 epochs* of training with real data (0.5235), a $\sim 9\times$ cost-reduction per trained model.



(a) CIFAR10 inner-loop training



(b) CIFAR10 GTN samples



(c) CIFAR10 correlation

Figure 4: Teaching CIFAR10 with GTN-generated images. (a) CIFAR10 training set performance of the final learner (after 1,700 meta-optimization steps) across inner-loop learning iterations. (b) Samples generated by GTN to teach CIFAR10 are unrecognizable, despite being effective for training. Each column contains a different class, and each row is taken from the same inner-loop iteration (evenly spaced from all 128 iterations, early iterations at the top). (c) Correlation between performance prediction using GTN-data vs. Real Data. When considering the top half of architectures (as ranked by GTN evaluation), correlation between GTN evaluation and search evaluation is strong (0.5582 rank-correlation), suggesting that GTN-NAS has potential to uncover high performing architectures at a significantly lower cost. Architectures shown are uniformly sampled from the NAS search space. The top 10% of architectures according to the GTN evaluation (blue squares)– those likely to be selected by GTN-NAS–have high true performance.

Architecture search methods are composed of several semi-independent components, such as the choice of search space, search algorithm, and proxy evaluation of candidate architectures. GTNs are proposed as an improvement to this last component, i.e. as a new way to quickly evaluate a new architecture. Thus we test our method under the standard search space for CIFAR10, using a simple form of search (random search) for which there are previous benchmark results. In particular, we ran an architecture search experiment where we evaluated 800 randomly generated architectures trained with GTN-synthetic data. We present the performance after *final evaluation* of the best architecture found in Table 1. This experimental setting is similar to that of Zhang et al. (2018). Highlighting the potential of GTNs as an improved proxy evaluation for architectures, we achieve state-of-the-art results when controlling for search algorithm (the choice of which is orthogonal to our contribution). While it is an apples-to-oranges comparison, GTN-NAS is competitive even with methods that use more advanced search techniques than random search to propose architectures (Appendix E). GTN is compatible with such techniques, and would likely improve their performance, an interesting area of future work. Furthermore, because of the NAS search space, the modules GTN found can be used to create even larger networks. A further test of whether GTNs predictions generalize is if such larger networks would continue performing better than architectures generated by the real-data control, similarly scaled. We tried F=128 and show it indeed does perform better (Table 1), suggesting additional gains can be had by searching post-hoc for the correct F and N settings.

4 Discussion and Future Work

The results presented here suggest potential future applications and extensions of GTNs. Given the ability of GTNs to rapidly train new models, they are particularly useful when training many independent models is required (as we showed for NAS). Another such application would be to teach networks on demand to realize particular trade-offs between e.g. accuracy, inference time, and memory requirements. While to address a range of such trade-offs would ordinarily require training many models ahead of time and selecting amongst them (Elsken et al., 2019), GTNs could instead rapidly train a new network only when a particular trade-off is needed. Similarly, agents with unique combinations of skills could be created on demand when needed.

Interesting questions are raised by the lack of similarity between the synthetic GTN data and real MNIST and CIFAR10 data. That unrealistic and/or unrecognizable images can meaningfully affect NNs is reminiscent of the finding that deep neural networks are easily fooled by unrecognizable images (Nguyen et al., 2015). It is possible that if neural network architectures were functionally more similar to human brains, GTNs' synthetic data might more resemble real data. However, an

Table 1: Performance of different architecture search methods. Our results report mean $\pm$ SD of 5 evaluations of the same architecture with different initializations. It is common to report scores with and without Cutout (DeVries & Taylor, 2017), a data augmentation technique used during training. We found better architectures compared to other methods that reduce architecture evaluation speed and were tested with random search (Random Search+WS and Random Search+GHN). Increasing the width of the architecture found (F=128) further improves performance. Because each NAS method finds a different architecture, the number of parameters differs. Each method ran once.

Model	Error(%)	#params	GPU Days
Random Search + GHN (Zhang et al., 2018)	4.3 ± 0.1	5.1M	0.42
Random Search + Weight Sharing (Luo et al., 2018)	3.92	3.9M	0.25
Random Search + Real Data (baseline)	3.88 ± 0.08	12.4M	10
Random Search + GTN (ours)	**3.84** ± 0.06	8.2M	0.67
Random Search + Real Data + Cutout (baseline)	3.02 ± 0.03	12.4M	10
Random Search + GTN + Cutout (ours)	**2.92** ± 0.06	8.2M	0.67
Random Search + Real Data + Cutout (F=128) (baseline)	2.51 ± 0.13	151.7M	10
Random Search + GTN + Cutout (F=128) (ours)	**2.42** ± 0.03	97.9M	0.67

alternate (speculative) hypothesis is that the human brain might also be able to rapidly learn an arbitrary skill by being shown unnatural, unrecognizable data (recalling the novel Snow Crash).

The improved stability of training GTNs from weight normalization naturally suggests the hypothesis that weight normalization might similarly stabilize, and thus meaningfully improve, any techniques based on meta-gradients (e.g. MAML (Finn et al., 2017), learned optimizers (Metz et al., 2019), and learned update rules (Metz et al., 2018)). In future work, we will more deeply investigate how consistently, and to what degree, this hypothesis holds.

Both weight sharing and GHNs can be combined with GTNs by using the shared weights or HyperNetwork for initialization of proposed learners and then fine-tuning on GTN-produced data. GTNs could also be combined with more intelligent ways to propose which architecture to sample next such as NAO (Luo et al., 2018). Many other extensions would also be interesting to consider. GTNs could be trained for unsupervised learning, for example by training a useful embedding function. Additionally, they could be used to stabilize GAN training and prevent mode collapse (Appendix I shows encouraging initial results). One particularly promising extension is to introduce a closed-loop curriculum (i.e. one that responds dynamically to the performance of the learner throughout training), which we believe could significantly improve performance. For example, a recurrent GTN that is conditioned on previous learner outputs could adapt its samples to be appropriately easier or more difficult depending on an agent's learning progress, similar in spirit to the approach of a human tutor. Such closed-loop teaching can improve learning (Fan et al., 2018).

An additional interesting direction is having GTNs generate training environments for RL agents. Appendix H shows this works for the simple RL task of CartPole. That could be either for a predefined target task, or could be combined with more open-ended algorithms that attempt to continuously generate new, different, interesting tasks that foster learning (Clune, 2019; Wang et al., 2019a). Because GTNs can encode any possible environment, they (or something similar) may be necessary to have truly unconstrained, open-ended algorithms (Stanley et al., 2017). If techniques could be invented to coax GTNs to produce recognizable, human-meaningful training environments, the technique could also produce interesting virtual worlds for us to learn in, play in, or explore.

5 Conclusion

This paper introduces a new method called Generative Teaching Networks, wherein data generators are trained to produce effective training data through meta-learning. We have shown that such an approach can produce supervised datasets that yield better training performance than an equivalent amount of real training data, and generalize across architectures and random initializations. We leverage such efficient training data to create a fast NAS method that generates state-of-the-art architectures (controlling for the search algorithm). While GTNs may be of particular interest to the field of architecture search (where the computational cost to evaluate candidate architectures often limits the scope of its application), we believe that GTNs open up an intriguing and challenging line of research into a variety of algorithms that learn to generate their own training data.

REFERENCES

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*, 2018.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Jeff Clune. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence. *arXiv preprint arXiv:1905.10985*, 2019.

Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *International Conference on Learning Representations*, 2019.

Yang Fan, Fei Tian, Tao Qin, Xiang-Yang Li, and Tie-Yan Liu. Learning to teach. *arXiv preprint arXiv:1805.03643*, 2018.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1126–1135. JMLR. org, 2017.

JL Fleiss. Review papers: The statistical basis of meta-analysis. *Statistical methods in medical research*, 2(2):121–145, 1993.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pp. 2672–2680, 2014.

Alex Graves, Marc G. Bellemare, Jacob Menick, Rémi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pp. 1311–1320, 2017.

Andreas Griewank and Andrea Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015.

Rein Houthooft, Yuhua Chen, Phillip Isola, Bradly Stadie, Filip Wolski, OpenAI Jonathan Ho, and Pieter Abbeel. Evolved policy gradients. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 31*, pp. 5405–5414. Curran Associates, Inc., 2018.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Ksenia Konyushkova, Raphael Sznitman, and Pascal Fua. Learning active learning from data. In *NIPS*, 2017.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 19–34, 2018a.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018b.

Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in neural information processing systems*, pp. 7816–7827, 2018.

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, pp. 3, 2013.

Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 2113–2122. JMLR.org, 2015.

Luke Metz, Niru Maheswaranathan, Brian Cheung, and Jascha Sohl-Dickstein. Meta-learning update rules for unsupervised representation learning. *arXiv preprint arXiv:1804.00222*, 2018.

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-dickstein. Learned optimizers that outperform on wall-clock and validation loss, 2019.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *In Computer Vision and Pattern Recognition (CVPR '15)*, 2015.

Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. 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. 4095–4104, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 4780–4789, 2019.

Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pp. 901–909, 2016.

Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *NIPS*, 2016.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. In *International Conference on Learning Representations*, 2018.

Burr Settles. Active learning literature survey. Technical report, 2010.

Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150, 1995.

Akash Srivastava, Lazar Valkov, Chris Russell, Michael U. Gutmann, and Charles Sutton. Veegan: Reducing mode collapse in gans using implicit variational learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 30*, pp. 3308–3318. Curran Associates, Inc., 2017.

Kenneth O. Stanley, Joel Lehman, and Lisa Soros. Open-endedness: The last grand challenge youve never heard of. *O'Reilly Online*, 2017. URL `https://www.oreilly.com/ideas/open-endedness-the-last-grand-challenge-youve-never-heard-of`.

Ivor Tsang, James Kwok, and Pak-Ming Cheung. Core vector machines: Fast svm training on very large data sets. *Journal of Machine Learning Research*, 6:363–392, 04 2005.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019a.

Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A. Efros. Dataset distillation, 2019b.

Chris Zhang, Mengye Ren, and Raquel Urtasun. Graph hypernetworks for neural architecture search. *arXiv preprint arXiv:1810.05749*, 2018.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017. URL `https://arxiv.org/abs/1611.01578`.

Appendix A MNIST Experiment details

For the GTN training for MNIST we used a small conv-fc based architectures. This architecture had 2 conv layers with number of filters sampled from $U([32, 128])$ and $U([64, 256])$, respectively. After each conv layer we used a max pooling layer for dimensionality reduction. After the last conv layer we used a fully-connected layer with number of filters sampled from $U([64, 256])$. We used Kaiming Normal initialization (He et al., 2015) and LeakyReLUs (Maas et al., 2013) (with $\alpha = 0.1$). We use BatchNorm (Ioffe & Szegedy, 2015) for both the generator and the learners. The BatchNorm momentum for the learner was set to 0 (meta-training consistently converged to small values and we saw no significant gain from learning the value). The generator used consisted of 2 FC layers (1024 and $128 * H/4 * H/4$ filters, respectively, where H is the final width of the synthetic image). After the last FC layer we use 2 conv layers. The first conv had 64 filters. The second conv had 1 filter for MNIST and 3 for CIFAR10 and was followed by a Tanh to create the synthetic image. We found particularly important to normalize (mean of zero and variance of one) all datasets. Hyperparameters used can be found in Table 2.

Hyperparameter	Value
Learning Rate	0.01
Initial LR	0.02
Initial Momentum	0.5
Adam Beta_1	0.9
Adam Beta_2	0.999
Size of latent variable	128
Inner-Loop Batch Size	128
Outer-Loop Batch Size	128

Table 2: Hyperparameters for MNIST experiments

Appendix B CIFAR10 training details

For GTN training for CIFAR-10, we used a small learner with 5 convolutional layers followed by a global average pooling and an FC layer. The second and fourth convolution had stride=2 for dimensionality reduction. The number of filter of the first conv layer was sampled from $U([32, 128])$ while all others were sampled from $U([64, 256])$. Other details including the generator architecture used were the same as the MNIST experiments. Hyperparameters used can be found in Table 3. For CIFAR10 we augmented the real training set when training GTNs with random crop and horizontal flip. We do not add weight normalization to the final reported architectures, but we do so when we train architectures with GTN-generated data.

Hyperparameter	Value
Learning Rate	0.002
Initial LR	0.02
Initial Momentum	0.5
Adam Beta_1	0.9
Adam Beta_2	0.9
Adam ϵ	1e-5
Size of latent variable	128
Inner Loop Batch Size	128
Outer-Loop Batch Size	256

Table 3: Hyperparameters for CIFAR10 experiments

Appendix C Cell Search Space

When searching for the operations in a CNN cell, the 11 possible operations are listed below.

- identity
- 1×1 convolution
- 3×3 convolution
- $1 \times 3 + 3 \times 1$ convolution
- $1 \times 7 + 7 \times 1$ convolution
- 2×2 max pooling
- 3×3 max pooling
- 5×5 max pooling
- 2×2 average pooling
- 3×3 average pooling
- 5×5 average pooling

APPENDIX D COMPUTATION AND MEMORY COMPLEXITY

With the traditional training of DNNs with back-propagation, the memory requirements are proportional to the size of the network because activations during the forward propagation have to be stored for the backward propagation step. With meta-gradients, the memory requirement also grows with the number of inner-loop steps because all activations and weights have to be stored for the 2nd order gradient to be computed. This becomes impractical for large networks and/or many inner-loop steps. To reduce the memory requirements, we utilize gradient-checkpointing (Griewank & Walther, 2000) by only storing the computed weights of learner after each inner-loop step and re-computing the activations during the backward pass. This trick allows us to compute meta-gradients for networks with 10s of millions of parameters over hundreds of inner-loop steps in a single GPU. While in theory the computational cost of computing meta-gradients with gradient-checkpointing is 4x larger than computing gradients (and 12x larger than forward propagation), in our experiments it is about 2.5x slower than gradients through backpropagation due to parallelism. We could further reduce the memory requirements by utilizing reversable hypergradients (Maclaurin et al., 2015), but, in our case, we were not constrained by the number of inner-loop steps we can store in memory.

APPENDIX E EXTENDED NAS RESULTS

In the limited computation regime (less than 1 day of computation), the best methods were, in order, GHN, ENAS, GTN, and NAONet with a mean error of 2.84%, 2.89%, 2.92%, and 2.93%, respectively. A 0.08% difference on CIFAR10 represents 8 out of the 10k test samples. For that reason, we consider all of these methods as state of the art. Note that out of the four, GTN is the only one relying on Random Search for architecture proposal.

APPENDIX F CONDITIONED GENERATOR VS. XY-GENERATOR

Our experiments in the main paper conditioned the generator to create data with given labels, by concatenating a one-hot encoded label to the input vector. We also explored an alternative approach where the generator itself produced a target probability distribution to label the data it generates. Because more information is encoded into a soft label than a one-hot encoded one, we expected an improved training set to be generated by this variant. Indeed, such a “dark knowledge” distillation setup has been shown to perform better than learning from labels (Hinton et al., 2015). However, the results in Figure 5 indicate that jointly generating both images and their soft labels under-performs generating only images, although the result could change with different hyperparameter values and/or innovations that improve the stability of training.

APPENDIX G GTN GENERATES (SEEMINGLY) ENDLESS DATA

While optimizing images directly (i.e. optimizing a fixed tensor of images) would result in a fixed number of samples, optimizing a generator can potentially result in an unlimited amount of new

Table 4: Performance of different architecture search methods. Search with our method required 16h total, of which 8h were spent training the GTN and 8h were spent evaluating 800 architectures with GTN-produced synthetic data. Our results report mean ± SD of 5 evaluations of the same architecture with different initializations. It is common to report scores with and without Cutout (DeVries & Taylor, 2017), a data augmentation technique used during training.We found better architectures compared to other methods using random search (Random-WS and GHN-Top) and are competitive with algorithms that benefit from more advanced search methods (e.g. NAONet and ENAS employ non-random architecture proposals for performance gains; GTNs could be combined with such non-random proposals, which would likely further improve performance). Increasing the width of the architecture found (F=128) further improves performance.

Model	Error(%)	#params	Random	GPU Days
NASNet-A (Zoph & Le, 2017)	3.41	3.3M	✗	2000
AmoebaNet-B + Cutout (Real et al., 2019)	2.13	34.9M	✗	3150
DARTS + Cutout (Liu et al., 2018b)	2.83	4.6M	✗	4
NAONet + Cutout (Luo et al., 2018)	2.48	10.6M	✗	200
NAONet-WS (Luo et al., 2018)	3.53	2.5M	✗	0.3
NAONet-WS + Cutout (Luo et al., 2018)	2.93	2.5M	✗	0.3
ENAS (Pham et al., 2018)	3.54	4.6M	✗	0.45
ENAS + Cutout (Pham et al., 2018)	2.89	4.6M	✗	0.45
GHN Top-Best + Cutout (Zhang et al., 2018)	2.84 ± 0.07	5.7M	✗	0.84
GHN Top (Zhang et al., 2018)	4.3 ± 0.1	5.1M	✓	0.42
Random-WS (Luo et al., 2018)	3.92	3.9M	✓	0.25
Random Search + Real Data (baseline)	3.88 ± 0.08	12.4M	✓	10
RS + Real Data + Cutout (baseline)	3.02 ± 0.03	12.4M	✓	10
RS + Real Data + Cutout (F=128) (baseline)	2.51 ± 0.13	151.7M	✓	10
Random Search + GTN (ours)	**3.84** ± 0.06	8.2M	✓	0.67
Random Search + GTN + Cutout (ours)	**2.92** ± 0.06	8.2M	✓	0.67
RS + GTN + Cutout (F=128) (ours)	**2.42** ± 0.03	97.9M	✓	0.67

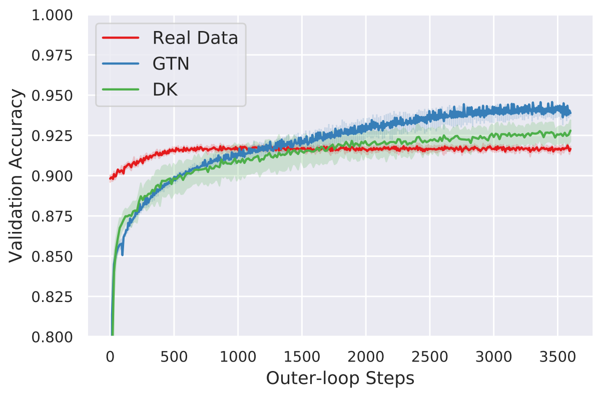
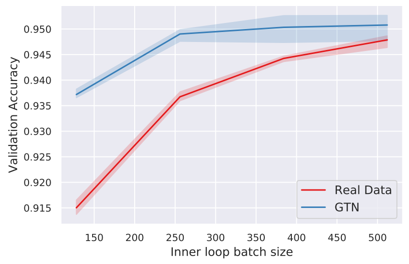
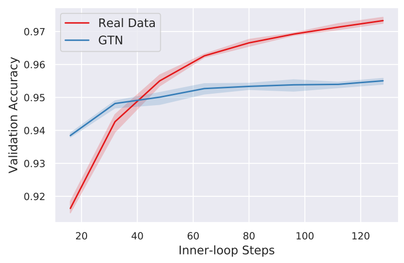


Figure 5: Comparison between a conditional generator and a generator that outputs an image/label pair. We expected the latter "dark knowledge" approach to outperform the conditional generator, but that does not seem to be the case. Because initialization and training of the dark knowledge variant were more sensitive, we believe a more rigorous tuning of the process could lead to a different result.

samples. We tested this generative capability by generating more data during evaluation (i.e. with no change to the meta-optimization procedure) in two ways. In the first experiment, we increase the amount of data in each inner-loop optimization step by increasing the batch size (which results in lower variance gradients). In the second experiment, we keep the number of samples per batch fixed, but increase the number of inner-loop optimization steps for which a new network is trained. Both cases result in an increased amount of training data. If the GTN generator has overfit to the number of inner-loop optimization steps during meta-training and/or the batch size, then we would not expect performance to improve when we have the generator produce more data. However, an alternate hypothesis is that the GTN is producing a healthy distribution of training data, irrespective of exactly how it is being used. Such a hypothesis would be supported by performance increase in these experiments.



(a) Increasing inner-loop batch size



(b) Increasing inner-loop optimization steps

Figure 6: (a) The left figure shows that even though GTN was meta-trained to generate synthetic data of batch size 128, sampling increasingly larger batches results in improved learner performance (the inner-loop optimization steps are fixed to 16). (b) The right figure shows that increasing the number of inner-loop optimization steps (beyond the 16 steps used during meta-training) improves learner performance. The performance gain with real data is larger in this setting. This improvement shows that GTNs do not overfit to a specific number of inner-loop optimization steps.

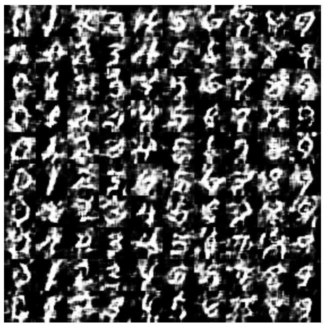


Figure 7: GTN samples w/o curriculum.

Figure 6a shows performance as a function of increasing batch size (beyond the batch size used during meta-optimization, i.e. 128). The increase in performance of GTN means that we can sample larger training sets from our generator (with diminishing returns) and that we are not limited by the choice of batch size during training (which is constrained due to both memory and computation requirements).

Figure 6b shows the results of generating more data by increasing the number of inner-loop optimization steps. Generalization to more inner-loop optimization steps is important when the number of inner-loop optimization steps used during meta-optimization is not enough to achieve maximum performance. This experiment also tests the generalization of the optimizer hyperparameters because they were optimized to maximize learner performance after a fixed number of steps. There is an increase in performance of the learner trained on GTN-generated data as the number of inner-loop optimization steps is increased, demonstrating that the GTN is producing generally useful data instead of overfitting to the number of inner-loop optimization steps during training (Figure 6b). Extending the conclusion from Figure 3b, in the very low data regime, GTN is significantly better than training on real data ($p < 0.05$). However, as more inner-loop optimization steps are taken and thus more unique data is available to the learner, training on the real data becomes more effective than learning from synthetic data ($p < 0.05$) (see Figure 6b).

Another interesting test for our generative model is to test the distribution of learners after they have trained on the synthetic data. We want to know, for instance, if training on synthetic samples from one GTN results in a functionally similar set of learner weights regardless of learner initialization

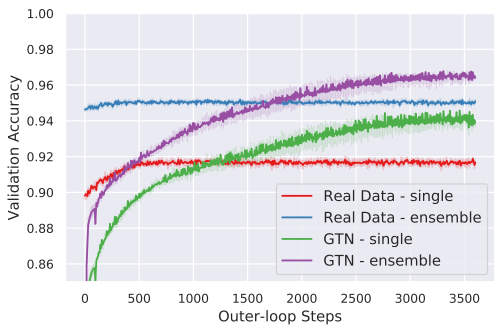


Figure 8: Performance of an ensemble of GTN learners vs. individual GTN learners. Ensembling a set of neural networks that each had different weight initializations, but were trained on data from the *same GTN* substantially improves performance. This result provides more evidence that GTNs generate a healthy distribution of training data and are not somehow forcing the learners to all learn a functionally equivalent solution.

(this phenomena can be called learner mode collapse). Learner mode collapse would prevent the performance gains that can be achieved through ensembling diverse learners. We tested for learner mode collapse by evaluating the performance (on held-out data and held-out architecture) of an ensemble of 32 randomly initialized learners that are trained on independent batches from the *same GTN*. To construct the ensemble, we average the predicted probability distributions across the learners to compute a combined prediction and accuracy. The results of this experiment can be seen in Figure 8, which shows that the combined performance of an ensemble is better (on average) than an individual learner, providing additional evidence that the distribution of synthetic data is healthy and allows ensembles to be harnessed to improve performance, as is standard with networks trained on real data.

APPENDIX H GTN FOR RL

To demonstrate the potential of GTNs for RL, we tested our approach with a small experiment on the classic CartPole test problem (see Brockman et al. (2016) for details on the domain. We conducted this experiment before the discovery that weight normalization improves GTN training, so these experiments do not feature it; it might further improve performance. For this experiment, the meta-objective the GTN is trained with is the advantage actor-critic formulation: $\log \pi(a|\theta_\pi)(R - V(s;\theta_v))$ (Mnih et al., 2016). The state-value V is provided by a separate neural network trained to estimate the average state-value for the learners produced so far during meta-training. The learners train on synthetic data via a single-step of SGD with a batch size of 512 and a mean squared error regression loss, meaning the inner loop is supervised learning. The outer-loop is reinforced because the simulator is non-differentiable. We could have also used an RL algorithm in the inner loop. In that scenario the GTN would have to learn to produce an entire synthetic world an RL agent would learn in. Thus, it would create the initial state and then iteratively receive actions and generate the next state and optionally a reward. For example, a GTN could learn to produce an entire MDP that an agent trains on, with the meta-objective being that the trained agent then performs well on a target task. We consider such synthetic (PO)MDPs an exciting direction for future research.

The score on CartPole is the number of frames out of 200 for which the pole is elevated. Both GTN and an A2C (Mnih et al., 2016) control effectively solve the problem (Figure 9). Interestingly, training GTNs takes the same number of simulator steps as training a single learner with policy-gradients (Figure 9). Incredibly, however, once trained, the synthetic data from a GTN can be used to train a learner to maximum performance in a single SGD step! While that is unlikely to be true for harder target RL tasks, these results suggest that the speed-up for architecture search from using GTNs in the RL domain can be even greater than in supervised domain.

The CartPole experiments feature a single-layer neural network with 64 hidden units and a tanh activation function for both the policy and the value network. The inner-loop batch size was 512

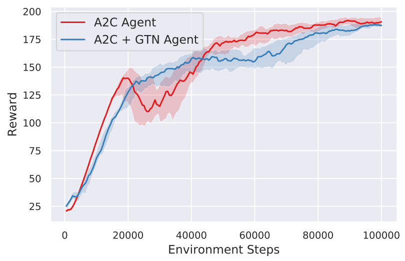


Figure 9: An A2C Agent control trains a single policy throughout all of training, while the GTN method optimizes a new, randomly initialized network at each iteration with a single step of SGD. With GTNs, we can therefore train many new, high-performing agents quickly. That would be useful in many ways, such as greatly accelerating architecture search algorithms.

and the number of inner-loop training iterations was 1. The observation space of this environment consists of a real-valued vector of size 4 (Cart position, Cart velocity, Pole position, Pole velocity). The action space consists of 2 discrete actions (move left or move right).

Appendix I Solving mode collapse in GANs with GTNs

We created an implementation of generative adversarial networks (GANs) (Goodfellow et al., 2014) and found they tend to generate the same class of images (e.g. only 1s, Figure 10), which is a common training pathology in GANs known as mode collapse (Srivastava et al., 2017). While there are techniques to prevent mode collapse (e.g. minibatch discrimination and historical averaging (Salimans et al., 2016)), we hypothesized that combining the ideas behind GTNs and GANs might provide a different, additional technique to help combat mode collapse. The idea is to add a discriminator to the GTN forcing the data it generates to both be realistic and help a learner perform well on the meta-objective of classifying MNIST. The reason this approach should help prevent mode collapse is that if the generator only produces one class of images, a learner trained on that data will not be able to classify all classes of images. This algorithm (GTN-GAN) was able to produce realistic images with no identifiable mode collapse (Figure 11). GTNs offer a different type of solution to the issue of mode collapse than the many that have been proposed, adding a new tool to our toolbox for solving that problem. Note we do not claim this approach is better than other techniques to prevent mode collapse, only that it is an interesting new type of option, perhaps one that could be productively combined with other techniques.

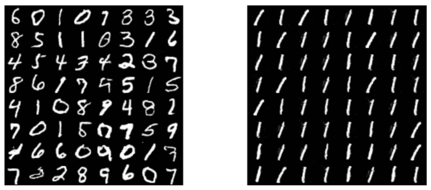


Figure 10: **Images generated by a basic GAN on MNIST before and after mode collapse.** The left image shows GAN-produced images early in GAN training and the right image shows GAN samples later in training after mode collapse has occurred due to training instabilities.

Appendix J Additional Motivation

There is an additional motivation for GTNs that involves long-term, ambitious research goals: GTN is a step towards algorithms that generate their own training environments, such that agents trained in them eventually solve tasks we otherwise do not know how to train agents to solve Clune (2019). It is important to pursue such algorithms because our capacity to conceive of effective training environments on our own as humans is limited, yet for our learning algorithms to achieve their full

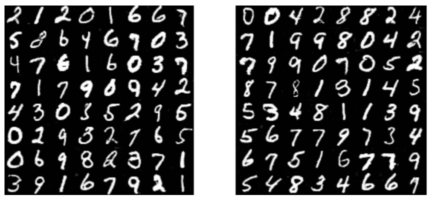


Figure 11: **Images generated by a GTN with an auxiliary GAN loss.** Combining GTNs with GANs produces far more realistic images than GTNs alone (which produced alien, unrecognizable images, Figure 7). The combination also stabilizes GAN training, preventing mode collapse.

potential they will ultimately need to consume vast and complex curricula of learning challenges and data. Algorithms for generating curricula, such as the the paired open-ended trailblazer (POET) algorithm Wang et al. (2019a), have proven effective for achieving behaviors that would otherwise be out of reach, but no algorithm yet can generate completely unconstrained training conditions. For example, POET searches for training environments within a highly restricted preconceived space of problems. GTNs are exciting because they can encode a rich set of possible environments with minimal assumptions, ranging from labeled data for supervised learning to (in theory) entire complex virtual RL domains (with their own learned internal physics). Because RNNs are Turing-complete Siegelmann & Sontag (1995), GTNs should be able to theoretically encode all possible learning environments. Of course, while what is theoretically possible is different from what is achievable in practice, GTNs give us an expressive environmental encoding to begin exploring what potential is unlocked when we can learn to generate sophisticated learning environments. The initial results presented here show that GTNs can be trained end-to-end with gradient descent through the entire learning process; such end-to-end learning has proven highly scalable before, and may similarly in the future enable learning expressive GTNs that encode complex learning environments.