Evaluating the Robustness of Generative Teaching Networks

Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth O. Stanley, Jeff Clune

Kurt Willis

July 4, 2021

Abstract

Training machine learning models from scratch is computationally demanding and efficient network training is desirable in order to reduce cost and time demands. Generative Teaching Networks, a meta-learning algorithm, have been proposed to rapidly train new architectures and can even be beneficial to Neural Architecture Search. This work is motivated by the question if possible downsides exist for training models in this fashion or if perhaps it comes with benefits by running robustness performance tests and comparing to traditionally trained models.

1 Introduction

Meta-learning is concerned about "learning to learn". In machine learning there are generally 3 components that come into play. The **environment**, the learner **model** and the learning **algorithm**. In traditional machine learning, the environment and the algorithm are seen as fixed. Only the model weights are adapted via the learning algorithm after receiving a signal from the environment (through the training data). Generative Teaching Networks (GTNs) (Such et al., 2019) aim to learn the environment and parameters of the learning algorithm to accelerate the learning process for a variety of different kinds of models. The environment and algorithm parameters are learned via meta-learning by maximizing the performance of models trained via GTNs on the training data.

A *Generative Teaching Network* is a generative model that is able to generate images from a latent vector $\mathbf{z}$ coming from some latent distribution. One outer-loop cycle (meta training step) consists of a full network training cycle (64[1] inner-loop update steps).

[1]The workings of GTNs will be illustrated by presenting the specific hyperparameter values used in the experiments, although most of these can be set arbitrarily.

Algorithm Generative Teaching Networks

```
initialize $G_{\boldsymbol{\omega}}$
for 2000 times do
    sample $D_{\boldsymbol{\theta}}$
    for 64 times do
        $\mathbf{z} = (\mathbf{z}_x, \mathbf{z}_y) \sim$ sample randomly
        $\mathbf{x} \leftarrow G_{\boldsymbol{\omega}}(\mathbf{z})$
        $\hat{\mathbf{y}} \leftarrow D_{\boldsymbol{\theta}}(\mathbf{x})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \nabla_{\boldsymbol{\theta}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{z}_y)$
    end for
    $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \gamma \nabla_{\boldsymbol{\omega}} \mathcal{L}(D_{\boldsymbol{\theta}}(\mathbf{x}_{\texttt{train}}), \mathbf{y}_{\texttt{train}})$
end for
```

An inner-loop cycle consists of

1. Randomly sampling a latent vector $\mathbf{z} = (\mathbf{z}_x, \mathbf{z}_y) \in \mathbb{R}^{128+10}$ (consisting of a latent code $\mathbf{z}_x$ and a label $\mathbf{z}_y$).
2. Generating an image $\mathbf{x} = G_{\boldsymbol{\omega}}(\mathbf{z}) \in \mathbb{R}^{28\times 28}$ by passing the latent vector $\mathbf{z}$ to the generator $G_{\boldsymbol{\omega}}$.
3. Calculating the class probability vector (predicted label) $\hat{\mathbf{y}} = D_{\boldsymbol{\theta}}(\mathbf{x}) \in \mathbb{R}^{10}$ by passing the image $\mathbf{x}$ to the discriminator network $D_{\boldsymbol{\theta}}$.
4. Updating the weights of the discriminator $D_{\boldsymbol{\theta}}$ by evaluating the cross-entropy loss $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{z}_y)$ and computing the gradient with respect to θ.

The outer-loop performs the following steps.

1. Sampling a new discriminator model $D_{\boldsymbol{\theta}}$ and randomly initializing it.
2. Performing 64 inner-loop update steps on $D_{\boldsymbol{\theta}}$.
3. Updating the weights of the generator $G_{\boldsymbol{\omega}}$ by evaluating the final performance of $D_{\boldsymbol{\theta}}$ on the training set $(\mathbf{x}_{\texttt{train}}, \mathbf{y}_{\texttt{train}})$ and computing the gradient with respect to ω.

The signal obtained by evaluating the discriminator on the training set is back-propagated through all of the 64 inner update cycles. This signal can also be used to learn further differentiable hyperparameters, such as the optimizer learning rate. All quantities, ($\mathbf{z}$, $\mathbf{x}$, ...) are batched in mini-batches of size 128.

In the main setup: *full curriculum*, the latent vectors $\mathbf{z}$, the learning rates and momentum parameters are learned for each of the 64 update steps. When dealing with the MNIST dataset this can result in over 95% accuracy after a few update steps. By contrast, traditional, unoptimized ”vanilla” learning takes about 390 update steps to reach similar accuracies.

Although meta-learning itself comes with a lot of overhead, working with a trained GTN comes with a few benefits. The authors mention rapid training and efficient Nerual Architecture Search Such et al., 2019. This work will further examine trade-offs by running robustness performance tests for models trained with GTNs and these to models trained in a traditional manner.

However, first some general notes. Training custom architectures with a trained GTN did not work out of the box. The possible architecture choices are limited. For example, weight normalization is required and batch normalization does not work. The authors make note of this, however, pooling layers also don't seem to work. This means the tests are restricted to a small set of selected architectures. The authors seem to train the GTN on one fixed architecture only. The rate of success of transferring a trained GTN to another architecture is limited.

2 Robustness Performance Tests

The following performance tests only focus on the MNIST dataset, as the CIFAR10 dataset results were deemed too unreliable to make any significant claims. For all experiments, two different GTN models have been trained that were optimized for the `'base_larger'` and `'base_larger3'` models respectively. These two GTN choices each trained a total of 9 different architecture *learner types*. Furthermore, three different learning algorithms are considered, `'10_gtn'`, `'gtn'` and `'vanilla'`. Models trained with `'10_gtn'` received 10 (inner loop) update steps from the GTN, `'gtn'` the full 64 steps. Models trained with `'vanilla'` are trained with a classical learning algorithm (Adam optimizer, 1 epoch over training data). In order to have a fair comparison, the criteria for accepting models of a certain learner type is that the minimum value over the learning algorithms (`'10_gtn'`, `'gtn'`, `'vanilla'`) has a mean accuracy greater than 0.7 for 5 individually trained models. This limited the choice of available learner types to only two (`'base_larger'` and *'base_larger3'*) of 9.

The robustness tests include input corruption by augmentation (adding noise and blurring) and the FGSM and LBFGS adversarial attacks. All tests are performed on unseen test data of size 10,000 and 1,000 for the augmentation and adversarial attacks respectively. In all tests, the input image is clipped to its valid range.

2.1 Noise Corruption

In this setting, Gaussian noise that is sampled from a normal distribution with standard deviation of β is added to the input.

$$\tilde{\mathbf{x}} = \mathrm{clip}[\,\mathbf{x} + \beta\mathbf{z}\,], \quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

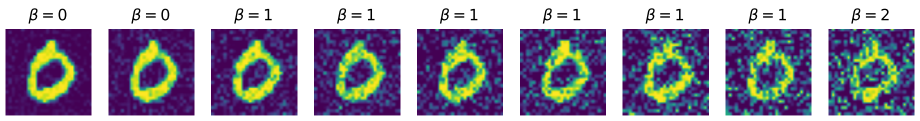


Figure 1: Noise corruption of varying strength β.

Figure 1 shows a randomly selected input example with varying noise strengths.

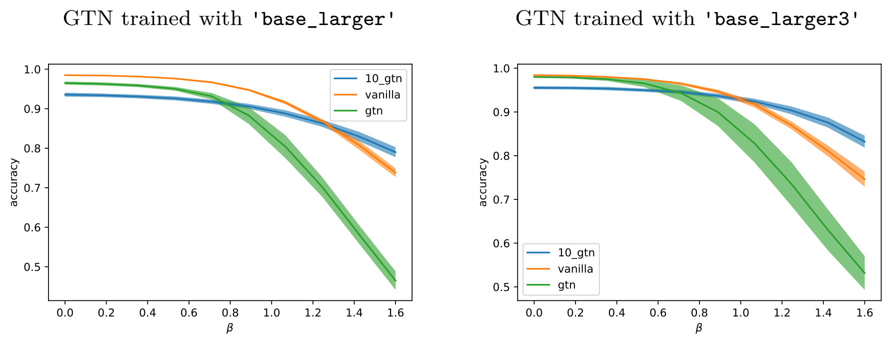


Figure 2: Accuracies of trained models under varying input noise corruption.

Figure 2 shows the results of the noise corruption performance tests.

2.2 Blurring

For this test, a blur filter is applied to the input via convolution with a Gaussian kernel with standard deviation of β.

$$\tilde{\mathbf{x}} = \text{gaussian\_blur}_\beta(\mathbf{x})$$

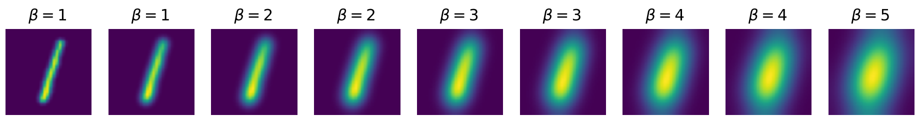
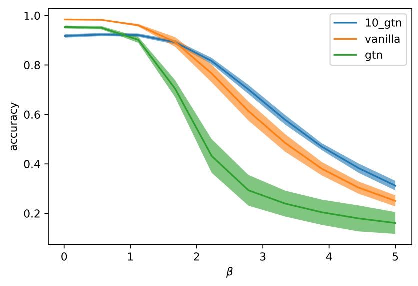


Figure 3: Gaussian-blur filter applied with varying Gaussian kernel std β.

GTN trained with `'base_larger'` GTN trained with `'base_larger3'`



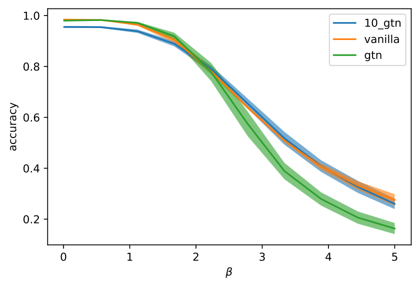


Figure 4: Accuracies of trained models under varying Gaussian-blur filters.

2.3 FGSM-Attack

The Fast-Gradient-Sign-Method is performed by calculating the gradient of the loss with respect to the input. The input is then moved by β in the direction (sign) of increasing loss. This is done for 30 steps. The formula for the update step is given in eq. (1). Figure 5 shows the outcome of varying β strengths on a random sample.

$$\begin{aligned} \mathbf{x} &\leftarrow \mathbf{x} + \beta\, \mathrm{sign}\left(\nabla_{\mathbf{x}} \mathcal{L}(D(\mathbf{x}), y)\right) \\ \mathbf{x} &\leftarrow \mathrm{clip}[\,\mathbf{x}\,] \end{aligned} \tag{1}$$

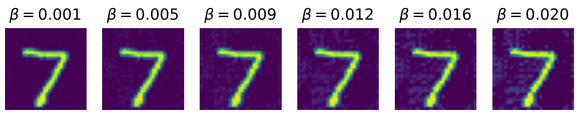


Figure 5: Results after 30 steps of FGSM-attack with varying β.

GTN trained with `'base_larger'` GTN trained with `'base_larger3'`

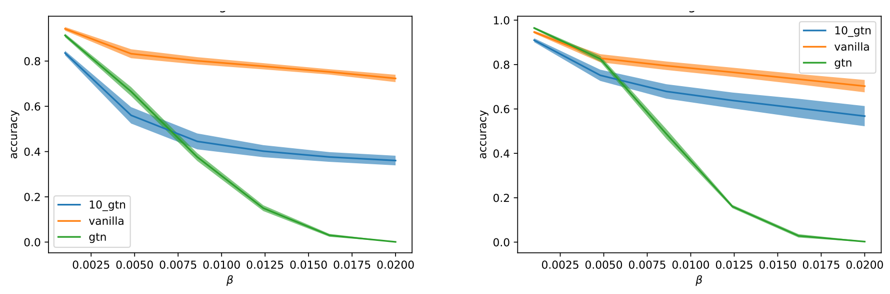


Figure 6: Accuracies of trained models under FGSM-attack with varying strengths.

2.4 LBFGS

The Box-constrained LBFGS-attack is performed similarly to the FGSM-attack by calculating the gradient of the loss with respect to the input. The loss here, however, is the cross-entropy loss minus the $l2$ distance of the current input to the original input multiplied by $\frac{1}{\beta}$ (this ensures that the computed adversarial output stays close to the original sample). The input is then moved by the update rate $\varepsilon = 0.002$ times the gradient of the loss. This is also done for 30 steps. The formula for the update step is given in eq. (1). Figure 7 shows the outcome of varying β strengths on a random sample.

$$\begin{aligned} \mathbf{x} &\leftarrow \mathbf{x} + \varepsilon \left(\nabla_{\mathbf{x}} \mathcal{L}(D(\mathbf{x}), y) - \frac{1}{\beta} \nabla_{\mathbf{x}} \|\mathbf{x} - \mathbf{x}^0\|_2 \right) \\ \mathbf{x} &\leftarrow \text{clip}[\,\mathbf{x}\,] \end{aligned} \tag{2}$$

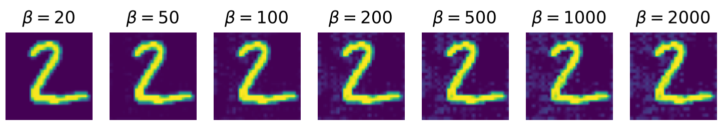


Figure 7: Results after 30 steps of LBFGS-attack with varying β.

GTN trained with `'base_larger'` GTN trained with `'base_larger3'`

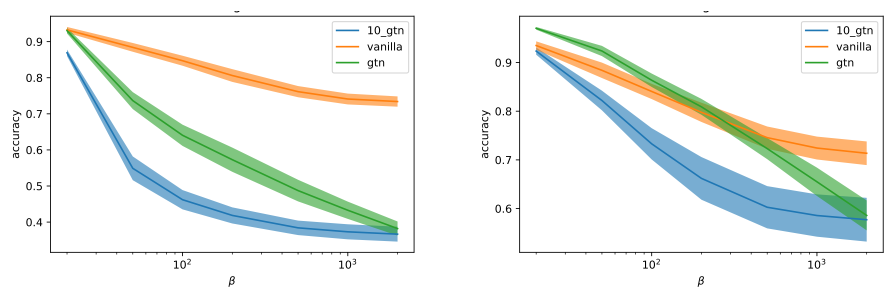


Figure 8: Accuracies of trained models under LBFGS-attack with varying strengths.

3 Conclusion

The general narrative has been that training a network to higher accuracies leads to decreasing generalization (Zhang et al., 2019). This is verified by comparing the networks trained with GTNs for the full 64 steps with those trained for only 10 steps. However, networks trained with a vanilla learning algorithm seem to, for the most part, outperform those trained by GTNs while still showing increased robustness towards input corruption and adversarial attacks. This leads to the conclusion that networks trained with GTNs are indeed less robust compared to those trained in a traditional way. However, since these experiments are only run on one dataset and GTNs will likely improve over time, more data is required to make any substantial claims.

References

[1] Felipe Petroski Such et al. *Generative Teaching Networks: Accelerating Neural Architecture Search by Learning to Generate Synthetic Training Data*. 2019. arXiv: `1912.07768 [cs.LG]`.

[2] Hongyang Zhang et al. *Theoretically Principled Trade-off between Robustness and Accuracy*. 2019. arXiv: `1901.08573 [cs.LG]`.