Transferring Learned Optimizers

Narek Daduryan
UCLA
COM SCI 260C Deep Learning

Abstract

Learned optimizers are able to outperform traditional optimizers on a variety of machine learning tasks. However, their high computational cost makes it difficult to be practically usable. The ability to transfer a learned optimizer may prove beneficial to offset their high initial cost. We examine the transferability of learned optimizers across different domains. While learned optimizers generally can be transferred to variations in model architecture, they may lose their competitive advantage compared to traditional optimizers. Large changes in models may mark the learned optimizers obsolete when transferred. However, we note that transferring learned optimizers across datasets, or with small architectural variations, may present an advantage in using learned optimizers.

1 Background

Machine learning models have traditionally used handcrafted optimization techniques derived from theory. The most fundamental optimization method has been gradient descent; updating model parameters directly based on the direction and magnitude of their gradients. Further advances in optimizations have all generally built upon gradient descent, including concepts such as momentum and normalization. Learned optimizers (Chen et al., 2021), on the other hand, view the optimization problem itself as a machine learning problem. This can allow the optimizer to learn novel update rules which as best suited for a certain optimization task.

Learned optimizers are generally viewed as sequence-to-sequence models, which take as input the parameter gradients (or some engineered feature-set based on the model parameters and gradients), and output model parameter updates at each time step. Learned optimizers have been shown to outperform traditional optimization techniques in both convergence speed and final model accuracy (Chen et al., 2021, 2020). However, using learned optimizers introduces a "meta-learning" stage, where the optimizer itself must be trained on the optimizee. This introduces new computational constraints both in the time taken to meta-train optimizers, and in the high memory requirement due to the unrolled training of the optimizee model (Metz et al., 2022).

Due to the high initial cost of training these optimizers, it is beneficial to be able to re-use a learned optimizer for several training iterations of the final model, perhaps with variations in either model architecture or dataset. This would make learned optimizers attractive, as their high initial cost would be offset by the quicker and more effective training while running perhaps hundreds of experiments and fine-tuning a specific model architecture or dataset. However, this requires that an optimizer meta-trained on a specific model and dataset, when transfered to a variation, maintains comparable performance and still outperforms traditional optimizers. In our research, we aim to discern the transferability of learned optimizers.

2 Learned Optimizers

For all our tests, we used an LSTM based learned optimizer. The model architecture can be found in the appendix. The LSTM based learned optimizer processes each parameter independently. During meta-training and usage of the optimizer, each individual parameter of the target model is treated as the batch input to the optimizer. The optimizer trains only one set of weights for the entire model. Crucially, this means the optimizer is completely unaware of any cross-parameter relationships. We discuss this more in our discussion.

For training larger models, including the convolutional neural network and transformer neural network, we ensured to use a pre processing step for the input gradients (Andrychowicz et al., 2016). This step is one way to normalize the input gradients across the different parameters in the neural network. Since we are only training one optimizer model for all the model parameters, the optimizer must be robust enough on different extremes of input magnitudes. In our training, conv-net and transformer models were unable to have optimizers trained that would converge without the pre-processing step. The pre-processing step had negligible effect on small models such as the quadratic regression, as expected.

3 Model Results

For the following machine learning models, we implemented the models "from scratch" in PyTorch, by making direct use of PyTorch Parameters rather than using traditional high-level modules, in order to allow direct manipulation of parameters and their gradients - a step required for meta-learning. This follows (Andrychowicz et al., 2016); however, we were able to use native PyTorch modules as a base class, rather than creating a new "MetaModule" implementation for the classes.

Our methodology included training a learned optimizer on the model (which tuning hyper-parameters) for a number of epochs. We then used the learned optimizer to fully train the model for a larger number of epochs, and compared the results with traditional optimizers ran for the same number of epochs. All optimization graphs are averaged over a number of training runs to ensure significance (usually 100 tests).

3.1 Quadratic Functions

Following in the footsteps of the L2O primer (Chen et al., 2021), we decided to ensure the learned optimizers were working as intended on an oversimplified optimization problem. We learned a randomized ten-dimensional parameter for a quadratic model. Our results 1 followed the primer paper, and showcase the learned optimizer (denoted `LSTM` in the figure) outperforming traditional optimizers. It is able to achieve a much lower final loss after 100 training steps. The learned optimizer was meta-trained for 20 epochs, and all optimizers completed final training for 100 epochs.

3.2 MLP on MNIST

We proceeded to train learned optimizers for image datasets. Our first model was a simple fully connected neural network with one hidden layer (MLP). The model in our experiment had approximately 20k parameters. Since each parameter is ran through the optimizer model independently, the highly varying magnitudes of different parameters became an issue. We were unable to obtain competitive results without pre-processing the gradients. We implemented pre-processing (Andrychowicz et al., 2016), and meta-trained the learned optimizer for 20 epochs. All optimizers then fully trained the network for 100 epochs. We get comparable results to the primer paper (Chen et al., 2021). The learned converges approximately as well as the Nesterov momentum optimizer (`NAG`); however, the learned optimizer does seem to consistently optimize faster than `NAG`.

Figure 2 shows the training loss; the learned optimizer is able to train faster at the start of training. There are also two "spikes" that all optimizers seem to encounter; however, the learned model seems to not spike up in loss, but rather is able to lower the loss sooner during these spikes. This may imply that our learned optimizer is more robust; however, in this instance the learned optimizer does not seem to converge to a statistically-significant lower final loss when compared to `NAG`.

3.3 ConvNet on MNIST

Expanding on the experiments in the primer paper, we decided to create a convolutional neural network (ConvNet) to train on the MNIST dataset. The model architecture is placed in the appendix. The model has less total parameters than the fully connected network, but is deeper with two convolutional layers and an additional hidden layer.

The meta-optimizer was trained for 20 epochs, and all optimizers trained the final model for 100 epochs. Pre-processing was used in the learned optimizer to obtain competitive results. Figure 3 shows the training loss of the convolutional network (denoted `ResNet` in the figure). We again see the learned optimizer being competitive with the traditional optimizers. We again see the learned optimizer greatly outperforming other optimizers early in the training process. It converges much faster in the first 20 epochs. However, in this instance, the learned optimizer ends up with a slightly worse training loss compared to `Adam` (Curiously, `Adam` even beats `NAG` in this instance).

4 Transferability of Learned Optimizers

To our interest was how well a learned optimizer would perform on "out-of-distribution tasks", or on models it wasn't meta-trained on. We experi-

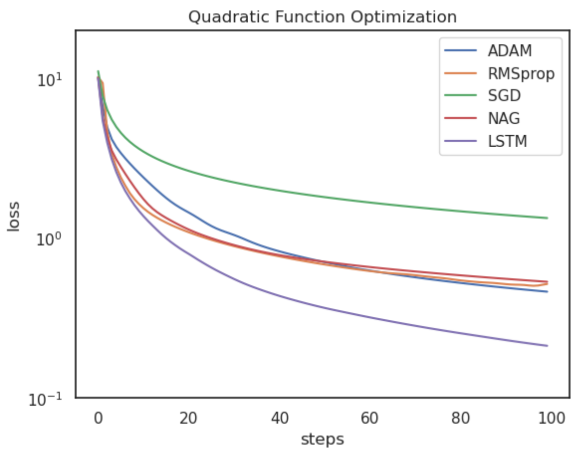

Figure 1: Training loss on quadratic model with several traditional optimizers, compared to learned (LSTM) optimizer.

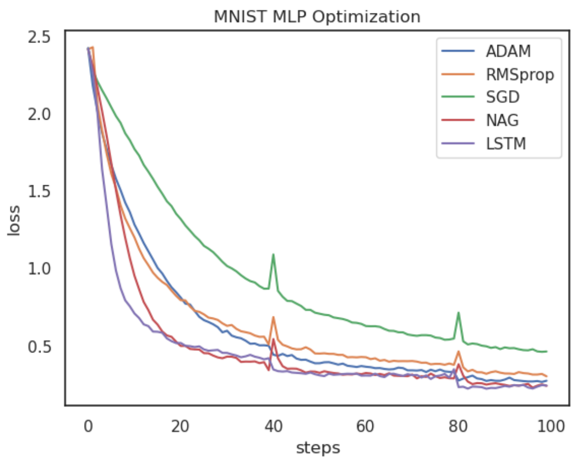

Figure 2: Training loss on a fully connected neural network (MLP) on MNIST dataset.

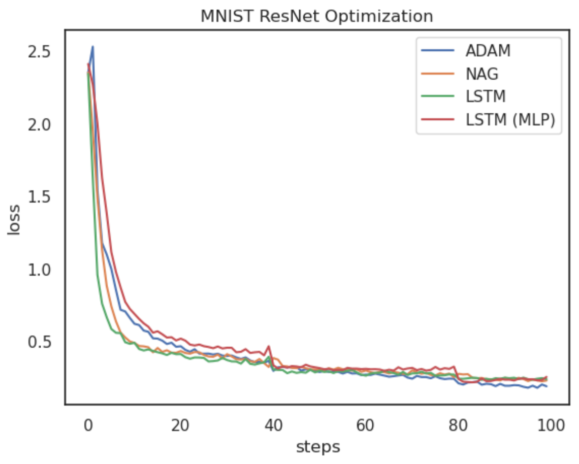

Figure 3: Training loss of convolutional neural network on MNIST dataset. Also shown: transfer of MLP optimizer to ConvNet

mented with changing both the model architecture, as well as the dataset, to examine how well learned-optimizers generalize.

4.1 Model Architecture Transferability

We took the MNIST dataset and experimented with making modifications both big and small to the model architecture. We then used the original learned optimizer to train the modified model.

We first tried making a large architectural change: swapping between a fully-connected neural network and convolutional neural network.

Figure 3 includes the `LSTM (MLP)` learned optimizer; this optimizer was trained only on the fully-connected (MLP) network on MNIST, and then used to train the ConvNet. As seen in the figure, the optimizer does perform competitively, but is worse than both the traditional optimizers and the ConNet learned optimizer.

Figure 4 showcases the opposite direction: we used a learned optimizer meta-trained on a ConvNet on MNIST, and used it to train a MLP on MNIST. Surprisingly, the transferability in this direction is greatly hindered. Although the ConvNet optimizer does seem to be able to converge the MLP, it occurs very slowly.

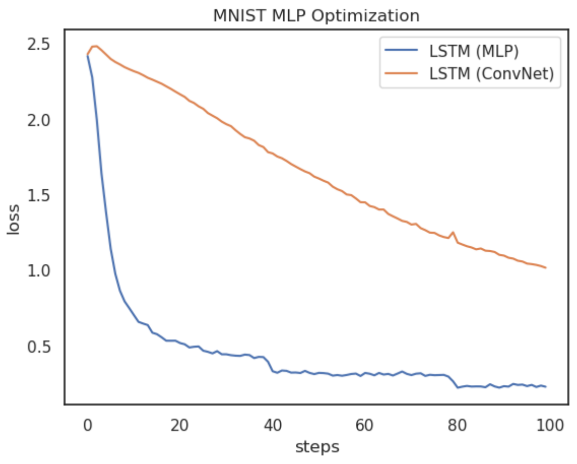

Figure 4: Training MLP on MNIST with a learned optimizer transferred from ConvNet meta-training.

We then wanted to experiment with making relatively minor changes to the ConvNet model architecture and examining the transferability of the original learned optimizer. We experimented with varying the activation function used in the output layer, and changing the convolution kernel size, both of which represent small variables that may be tweaked in hopes of improving model performance.

We meta-trained the learned optimizer originally on a ConvNet with kernel size 3x3 (for both layers). We created a new ConvNet model with a 5x5 kernel in both layers, and used the previously trained optimizer to train it. Figure 5 shows training performance with the transferred optimizer. We see the transferred optimizer is able to successfully train the modified model. It seems to maintain the same characteristics of the original architecture; it maintains superior performance within the first 20 steps, then performs equally well with `ADAM`.

The original learned optimizer was meta-trained on a ConvNet with final ReLU activation function. We created a new ConvNet with a Sigmoid activation, and tested the transferability of the original learned optimizer. We see the same result as above, where the transferred learned optimizer maintains the same characteristics of the original architecture, and is still competitive with the traditional optimizer.

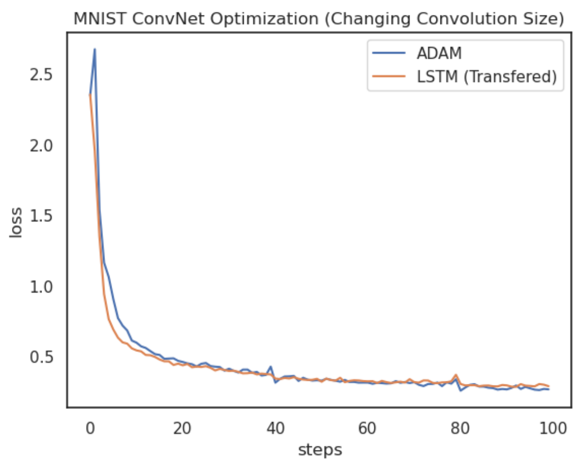

Figure 5: Training loss of ConvNet with kernel size 5x5 on MNIST; learned optimizer was meta-trained with a 3x3 convolution kernel.

4.2 Dataset Transferability

We experimented with transferring the dataset that we complete final training on. We first meta-trained a learned optimizer for ConvNet on one dataset, then compared its performance during training of a separate dataset. In our experiments, we switched between MNIST and FashionMNIST due to its plug-and-play nature. This will isolate the dataset as the only variable, as the model architecture does not change.

On ConvNet, we were able to maintain competitive results with our learned optimizer after transferring datasets. When transferring from MNIST to FashionMNIST, the learned optimizer transferred almost identically, maintaining its faster convergence at the start and eventually converging to the same end loss (Figure 7).

When examining the other direction (meta-training on FashionMNIST, and training on MNIST), we again see the optimizer exhibits transferability. However, in this case the transferred optimizer performs suboptimally compared to a traditional optimizer; it also loses its faster-convergence property within the first 20 steps (Figure 8).

Note: the above figures incorrectly state `MLP` rather than `ConvNet`.

5 Discussion

5.1 Learned Optimizers

Learned optimizers are able to achieve competitive results, sometimes surpassing traditional optimizers. Our experiments were able to reproduce the advantage of learned optimizers. Related works show further improvements in the performance of learned optimizers, with techniques such as cross-parameter inputs (Gärtner et al., 2023; Chen et al., 2022), further pre-processing (Shazeer and Stern, 2018), and modified optimizer architecture (Chen et al., 2022; Gärtner et al., 2023), etc.

In our experiments, we meta-trained optimizers for 20 steps, and validated them during final training of 100 steps. In virtually all the experiments, the learned optimizer, as expected, greatly outperforms traditional optimizers in the first 20 steps. However, they generally were unable to overwhelmingly outperform traditional optimizers after that cut off. This suggests more work may be needed in generalizing the optimizers to different training lengths, or meta-training optimizers for longer, which is less computationally feasible.

One of the largest hurdles in the usage of learned optimizers is their initial meta-training cost in wall-clock time. Although learned optimizers converge quicker in terms of steps or epochs, the real time performance is equally important. In our experiments, the final training using either traditional or learned optimizers was relatively comparable (although traditional optimizers did generally run each iteration quicker). However, the learned optimizers had a meta-training phase, which was an order of magnitude more expensive (both resources and wall-clock time) than final training. Thus, any time benefits of learned optimizers may be offset by the large initial cost.

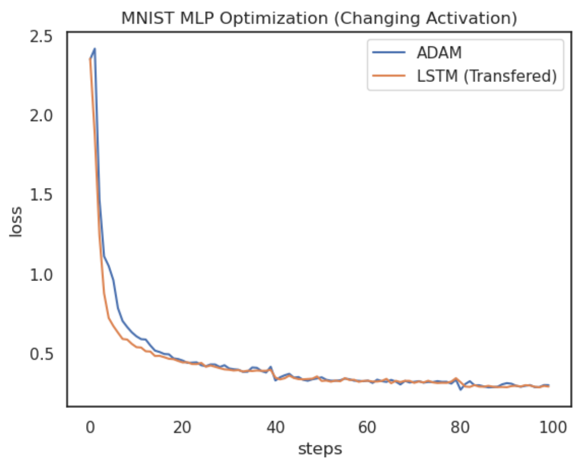

Figure 6: Training loss of ConvNet with Sigmoid activation on MNIST; learned optimizer was meta-trained with a ReLU activation function.

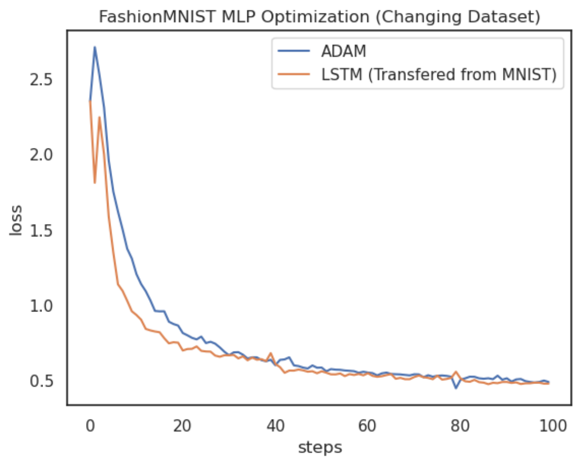

Figure 7: Training loss on ConvNet on FashionMNIST; learned optimizer was meta-trained on MNIST.

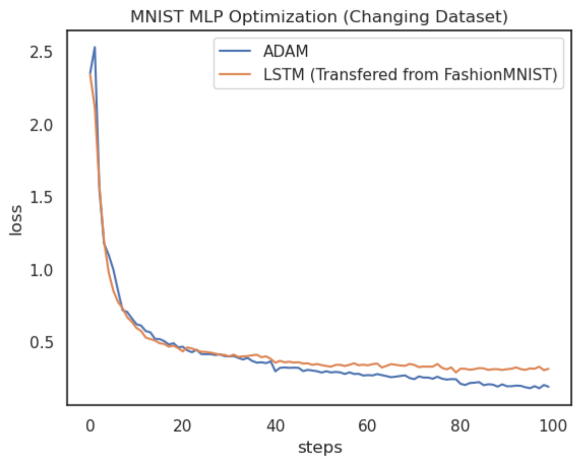

Figure 8: Training loss on ConvNet on MNIST; learned optimizer was meta-trained on FashionMNIST.

An active research field (e.g. (Metz et al., 2022)) is making learned optimizers both more efficient to meta-train, and faster during final training. This may prove to help make learned optimizers more practical.

5.2 Transferability

Our experiments showcase that learned optimizers can exhibit transferability in certain cases. When transferring model architectures, learned optimizers may be able to successfully train a model, and in some cases do so efficiently. However, the optimizers generally lose their competitveness in regards to convergence speed and final training loss when making too large of an architectural change. Our experiments showcased their feasibility when changing architectural variables such as activation functions or kernel sizes. Thus, they may not practically suitable for large variations in model architecture, but can present as useful when trying to quickly train multiple models with slight variations.

When transferring to different datasets on the same model architecture, the learned optimizers were generally able to maintain much better performance, as compared to different architectures. In our experiments, the learned optimizer may even maintain competitiveness compared to traditional optimizers, even after transferring datasets.

This suggests that learned optimizers may be successfully transferred to other datasets within the same model architecture. Since learned optimizers have a high initial meta-training cost, this transferability may present as one helpful avenue for practical use of learned optimizers. When experimenting with different datasets, e.g. during datset distillation or dataset augmentation, one may need to try training a model on variations of a dataset many times. If a learned optimizer can have superior performance to a traditional optimizer, and train faster, while transferring between variations of a dataset, it may be worth the one-time cost of meta-training a learned optimizer.

6 Further Research

This research was very insightful, and I hope to continue working on this project for my own interests.

I had originally also used learned optimizers on a small BERT transformer model, and intended to explore transferability between different language datasets or transformer architectures. However, the large computational cost proved to be a large hurdle in running experiments. I experimented with implementing various optimizations, including a low-rank parameter approximation (Chen et al., 2022), and got some success, but need to continue work in this area.

Another area of research is how to explicitly improve transferability. Just as we aim to improve generalizability in traditional machine learning models, we can "pre-train" a more general learned optimizer on a set of several model architectures or datasets. For example, a "transformer" learned optimizer, which is pre-trained on a dozen different common transformer architectures, and would work "out-of-the-box" with novel transformer architectures, still surpassing traditional optimizers. This would reduce the one-time cost hurdle, since users could use pre-trained optimizers which surpass traditional optimizers.

References

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. 2016. Learning to learn by gradient descent by gradient descent.

Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. 2021. Learning to optimize: A primer and a benchmark.

Tianlong Chen, Weiyi Zhang, Jingyang Zhou, Shiyu Chang, Sijia Liu, Lisa Amini, and Zhangyang Wang. 2020. Training stronger baselines for learning to optimize.

Xuxi Chen, Tianlong Chen, Yu Cheng, Weizhu Chen, Ahmed H. Awadallah, and Zhangyang Wang. 2022. Scalable learning to optimize: A learned optimizer can train big models. In *European Conference on Computer Vision (ECCV 2022)*.

Erik Gärtner, Luke Metz, Mykhaylo Andriluka, C. Daniel Freeman, and Cristian Sminchisescu. 2023. Transformer-based learned optimization.

Luke Metz, C. Daniel Freeman, James Harrison, Niru Maheswaranathan, and Jascha Sohl-Dickstein. 2022. Practical tradeoffs between memory, compute, and performance in learned optimizers.

Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost.

7 Appendix: Model Architectures

All codes can be found at `https://github.com/dadur604/learning-to-optimize`

The LSTM Learned optimizer has two LSTM cells, each with a hidden size of 20. The initial input is pre-processed gradients, and the output is a gradient update.