Review of Various Methods for Learning to Learn Using RNN

Apoorva Rastogi
Arizona State University
arasto17@asu.edu

Abstract

Meta-Learning or the term learning to learn is often talked about and used to describe the use of learning framework to learn on data of learning algorithm to achieve better results. In this report it is used in different forms for various different applications. Recurrent Neural Networks is a subclass of neural networks. In this report we see that they are highly versatile int their application and their inherent ability to store previous information is useful for variety of different application. We use Recurrent Neural Networks on black box optimization and also as an optimization algorithm for specific learning framework. Result shows us that it performs and generalizes quite well in both the scenarios.

1 Title and Authors

- Title - Learning to learn by gradient descent by gradient descent
  Authors - Marcin Andrychowicz, Misha Denil, Sergio Gómez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, Nando de Freitas
- Title - Learning to Learn without Gradient Descent by Gradient Descent.
  Authors - Chen, Yutian, Matthew W. Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Timothy P. Lillicrap, Matthew M. Botvinick and Nando de Freitas.

2 Introduction

A recurrent neural network (RNN) is the type of artificial neural network (ANN) . RNN remembers past inputs due to an internal memory which is useful for prediction tasks such as stock prices, generating text, transcriptions, and machine translation. In the traditional neural network, the inputs and the outputs are independent of each other, whereas the output in RNN is dependent on prior values within the sequence.

In this report we see Recurrent neural networks can be used in variety of ways. We see them being used as optimization algorithm for supervised learning. In [2] we even see them being used in black box optimization. Recurrent Neural Networks are used as supervised learning at meta-level to learn an algorithm for supervised learning.

Learning to learn can be used as both models and algorithms. In this sense learning to learn with neural networks blurs the classical distinction between models in algorithms. In the second paper the goal of meta learning is to produce an algorithm for black box optimization. Specifically they address the problem of finding a global minimizer of an unknown black box function f . The function f is not known but can be evaluated at an arbitrary point x.

Blackbox optimization (BBO) considers the design and analysis of algorithms for problems where the structure of the objective function and/or the constraints defining the set is unknown, exploitable, or non-existent.

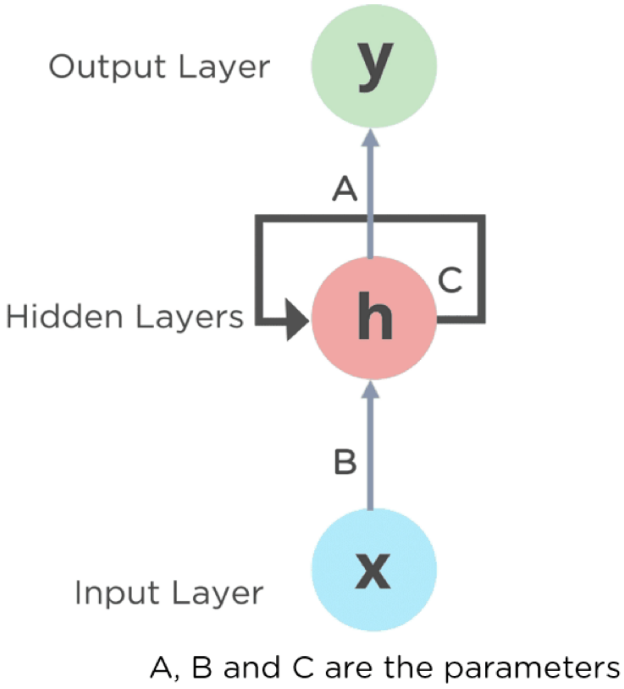

Figure 1: RNN Architecture

2.1 RNN

Neural networks try to imitate the synaptic response of the human brain to solve problems in the areas related to AI, machine learning, and deep learning, allowing computer programs to behave intelligently. RNNs are a type of neural network that can be used to model sequence data. To understand why we use RNNs in these two different setting we first need to understand the basic property of RNN and what makes it different from a traditional Artificial neural network . Recurrent Neural Networks differentiates from Multi-Layer Perceptrons (MLPs) in how information gets passed through the network. While MLPs pass information through the network independently without cycles, the RNN has cycles to transmit information back into itself. This enables them to take into account previous inputs $X_{0:t-1}$ and not only the current input X_t.

Here to represent RNN through mathematical functions[3] we denote the hidden state and the input at time step t respecively as $\mathbf{H}_t \in \mathbb{R}_{nh}$ and $\mathbf{X}_t \in \mathbb{R}_{nd}$ where n is number of samples, d is the number of inputs of each sample and h is the number of hidden units. Further, we use a weight matrix $\mathbf{W}_{xh} \in \mathbb{R}^{dh}$, hidden-state-to-hidden-state matrix $\mathbf{W}_{hh} \in \mathbb{R}^{hh}$ and a bias parameter $\mathbf{b}_h \in \mathbb{R}^{1h}$. Lastly, all these information get passed to a activation function ϕ which is usually a logistic, sigmoid or tanh function to prepair the gradients for usage in backpropagation. Putting all these notations together yields Equation 1 as the hidden variable.

$$\mathbf{H}_t = \phi_h(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h) \qquad (1)$$

The inputs and outputs in standard neural networks are independent of each another, however in some circumstances, it is necessary that the next output needs information about the previous input, such as when predicting the next word of a phrase. As a result, RNN was created, which used a Hidden Layer to overcome the problem. The most important component of RNN is the Hidden state, which remembers specific information about a sequence. RNNs have a Memory that stores all information about the calculations. It employs the same settings for each input since it produces the same outcome by performing the same task on all inputs or hidden layers. This property of RNN of able to retain and use memory makes it useful for various applications in Machine learning and AI. RNNs are a

type of neural network that has hidden states and allows past outputs to be used as inputs. However due to storing information for various iterations RNN are susceptible to vanishing/explding gradient.

2.1.1 LSTM

Vanishing/exploding gradient: The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers. In order to remedy the vanishing gradient problem, specific gates are used in some types of RNNs and usually have a well-defined purpose. LSTMs are a special kind of RNN — capable of learning long-term dependencies by remembering information for long periods is the default behavior. All RNN are in the form of a chain of repeating modules of a neural network. [4] provides us with a more in-depth understanding of LSTM. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer. LSTMs also have a chain-like structure, but the repeating module is a bit different structure. Instead of having a single neural network layer, four interacting layers are communicating extraordinarily. They have internal mechanisms called gates that can regulate the flow of information. These operations are used to allow the LSTM to keep or forget information. We have three different gates which are forget gate, input gate, and output gate that regulate information flow in an LSTM cell. LSTM enable RNN to learn what information is relevant to keep or forget during training.

2.1.2 DNC

A DNC is defined as a Differentiable neural computer.

To understand DNC in simple terms I borrow the explanation from [5]. DNC is introduced as a machine learning model, which consists of a neural network that can read from and write to an external memory matrix, analogous to the random-access memory in a conventional computer. Like a conventional computer, it can use its memory to represent and manipulate complex data structures, but, like a neural network, it can learn to do so from data. When trained with supervised learning, a DNC can successfully answer synthetic questions designed to emulate reasoning and inference problems in natural language. They show that it can learn tasks such as finding the shortest path between specified points and inferring the missing links in randomly generated graphs, and then generalize these tasks to specific graphs such as transport networks and family trees. When trained with reinforcement learning, a DNC can complete a moving blocks puzzle in which changing goals are specified by sequences of symbols. Taken together, our results demonstrate that DNCs have the capacity to solve complex, structured tasks that are inaccessible to neural networks without external read–write memory. We will see how we can use RNN for several ML Applications henceforth.

3 Problem Description

3.1 Using RNN as Learning Algorithm

Tasks in machine learning are often expressed as the problem of optimizing an objective function $f(\theta)$ defined over some domain . The goal in this case is to find the minimizer $\theta^* = argmin_{\theta \in \Theta} f(\theta)$ [1] . While any method capable of minimizing this objective function can be applied, the standard approach for differentiable functions is some form of gradient descent, resulting in a sequence of updates.

$$\theta_{t+1} = \theta_t - \alpha_t \nabla f(\theta_t) \qquad (2)$$

By training neural networks, we essentially mean we are minimizing a loss function. The value of this loss function gives us a measure how far from perfect is the performance of our network on a given dataset. The performance of vanilla gradient descent, however, is hampered by the fact that it only makes use of gradients and ignores second-order information. When using this form of vanilla gradient descent there are many issues listed as :

1. If gradient descent is not executed properly, it might lead to problems like vanishing gradient or exploding gradient problems. These problems occur when the gradient is too small or too large. And algorithms do not converge due to this.

2. There may be the issue of various minimal points. The lowest point is called global minimum, whereas rest of the points are called local minima. Our aim is to find global minima under the assumption that it gives us the performance on the minimizing the objective function
3. There is also a saddle point problem. This is a point in the data where the gradient is zero but is not an optimal point. We don't have a specific way to avoid this point and is still an active area of research.
4. We need to find the proper learning rate which helps us avoid these issue which will lead to algorithm being run multiple times.
5. This does not take into account the trajectory of function f which needs different learning rates at different point. Even if we do optimize to account for or to include different rates it is still based on specific parameters which means it is still hand designed and need further optimization
6. It applies same learning rate for different parameters and thus might not give optimum performance.

The equation of this formula does not take into account information specific to f and thus reforms same on every loss function which might not be an ideal solution. Various modifications to designs update rules such as including momentum etc. For example, in the deep learning community we have seen a proliferation of optimization methods specialized for high-dimensional, non-convex optimization problems. These include momentum Rprop, Adagrad, RMSprop, and ADAM. According to authors in [1], no general algorithm can perform better any random algorithm as it does not take into account the properties of that specific subclass of problems. They state No Free Lunch Theorem the reason for this and only a way to get a better performance it to get algorithm which takes into account the specifics of the particular problem.

In this paper they cast the problem of algorithm design as a learning problem. They use Neural Networks specifically Recurrent Neural Networks to design the algorithm for they wished to use its good generalization capability. We are aware of the ability of deep networks to generalize to new examples by learning interesting sub-structures. In [1] they aimed to leverage this generalization power, but also to lift it from simple supervised learning to the more general setting of optimization.

3.2 The other case is - Black Box Optimization

In [2] they address the problem of finding a global minimizer of an unknown (black-box) loss function f. That is, they wish to compute $x^* = argmin_{x \in X} f(x)$, where X is some search space of interest. The black-box function f is not available to the learner in simple closed form at test time, but can be evaluated at a query point x in the domain. In other words, we can only observe the function f through unbiased noisy point-wise observations y.

One of the most commonly used method for this is Bayesian optimization. Bayesian optimization is a sequential model-based approach to solving Black Box Optimization . A prior belief is prescribed over the possible objective functions and then sequentially refined as more data is observed. The Bayesian posterior represents updated beliefs on the likely objective function being optimized. Equipped with this probabilistic model, an acquisition function is sequentially induced that leverage the uncertainty in the posterior to guide exploration. Intuitively, the acquisition function evaluates the utility of candidate points for the next evaluation of f .

Bayesian Optimization builds a probability model of the objective function. Its framework has two key ingredients. Using samples from the true objective function a surrogate model is built (also called the response surface model) to approximate the true objective function, which consists of a prior distribution that captures beliefs about the behavior of the unknown objective function and an observation model that describes the data generation mechanism. The second ingredient is a selection function also known as acquisition function. Selection function defines the criteria by which the next query is chosen from the surrogate function [6]. After observing the output of each query of the objective, the prior is updated to produce a more informative posterior distribution over the space of objective functions.

In this paper, they present an approach for global optimization of black-box functions and contrast it with Bayesian optimization. In the meta-learning phase, they use a large number of differentiable

functions generated with a GP to train RNN optimizers by gradient descent. This problem is useful in many applications one of which is Hyperparameter Optimization for Machine Learning.

4 Methodology

4.1 RNN as Learning Algorithm

Here they develop a procedure for constructing a learning algorithm which performs well on a particular class of optimization problems. Casting algorithm design as a learning problem allows us to specify the class of problems we are interested in through example problem instances.

Generalization in this framework Generalization refers to the ability of our ML model to perform well on unseen data. Transfer Learning refers to ability of a model to perform well on a different data set other than its own but which somehow follows similar distribution or properties. To explain the importance of generalization in this case and how it corresponds to transfer learning the authors explains it as follows:

In ordinary statistical learning we have a particular function of interest, whose behavior is constrained through a data set of example function evaluations. In choosing a model we specify a set of inductive biases about how we think the function of interest should behave at points we have not observed, and generalization corresponds to the capacity to make predictions about the behavior of the target function at novel points. In our setting the examples are themselves problem instances, which means generalization corresponds to the ability to transfer knowledge between different problems. This reuse of problem structure is commonly known as transfer learning, and is often treated as a subject in its own right.

However, by taking a meta-learning perspective, they have cast the problem of transfer learning as one of generalization, which is much better studied in the machine learning community. One of the great success stories of deep-learning is that we can rely on the ability of deep networks to generalize to new examples by learning interesting sub-structures. In this work they leveraged this generalization power, but also to lift it from simple supervised learning to the more general setting of optimization.

4.1.1 Learning to Learn with Recurrent Neural networks

In [1] the optimizer is directly parameterized using ϕ. Update steps g_t are the output of a recurrent neural network m, parameterized by ϕ, whose state is denoted explicitly with h_t. for training the optimizer it will be convenient to have an objective that depends on the entire trajectory of optimization, for some horizon T, so it is defined as:

$$L(\phi) = \mathbb{E}_f \left[\sum_{t=1}^{T} w_t f(\theta_t) \right] \qquad (3)$$

$$\begin{bmatrix} g_t \\ h_{t+1} \end{bmatrix} = m(\nabla_t, h_t, \phi) \qquad (4)$$

$$\theta_{t+1} = \theta_t + g_t \qquad (5)$$

Here $w_t \in \mathbb{R}_{\geq 0}$ are arbitrary weights associated with each time-step and $\nabla_t = \nabla_\theta f(\theta_t)$. m is the recurrent neural network and its inputs are: (1.) ∇_t is the gradient of the function at time t. (2.) h_t is the previous state on neural network andn(3.) ϕ are the parameters of the network. The output g_t is used to compute the next value of θ

We can minimize the value of $L(\phi)$ using gradient descent on ϕ. The gradient estimate $\partial L(\phi)/\partial\phi$ can be computed by sampling a random function f and applying backpropagation to the computational graph in Figure 2. We allow gradients to flow along the solid edges in the graph, but gradients along the dashed edges are dropped. Ignoring gradients along the dashed edges amounts to making the assumption that the gradients of the optimizee do not depend on the optimizer parameters, i.e. $\partial\nabla_t/\partial\phi = 0$. This assumption allows us to avoid computing second derivatives of f.

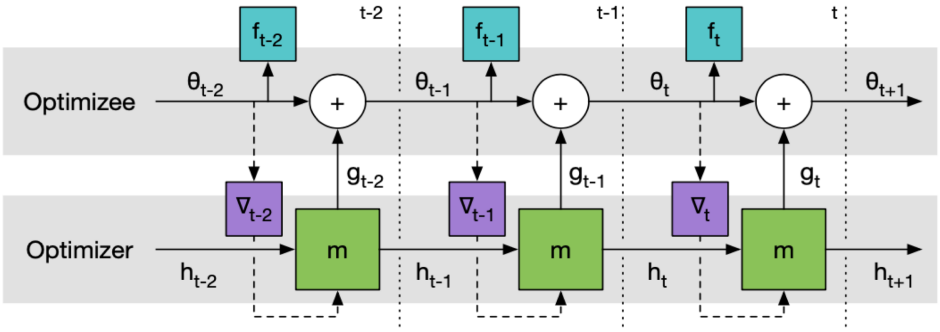

Figure 2: Computational graph used for computing the gradient of the optimizer.[1]

4.1.2 Coordinate wise LSTM Optimizer

To avoid optimizing tens of thousand of parameters they use an optimizer m which operates coordinatewise on the parameters of the objective function. This coordinatewise network architecture allows the use of a very small network that only looks at a single coordinate to define the optimizer and share optimizer parameters across different parameters of the optimizee. They implement the update rule for each coordinate using a two-layer Long Short Term Memory (LSTM) network. The network takes as input the optimizee gradient for a single coordinate as well as the previous hidden state and outputs the update for the corresponding optimizee parameter.This architecture is referred an LSTM optimizer. The use of recurrence allows the LSTM to learn dynamic update rules which integrate information from the history of gradients, similar to momentum. This is known to have many desirable properties in convex optimization.

4.2 Black-Box Optimization

In [2] they define:
A black-box optimization algorithm can be summarized by the following loop:

1. Given the current state of knowledge $\mathbf{h}_t$ propose a query point $\mathbf{x}_t$
2. Observe the response y_t.
3. Update any internal statistics to produce $\mathbf{h}_{t+1}$
   This easily maps onto the classical frameworks presented in the previous section where the update step computes statistics and the query step uses these statistics for exploration. In this work we take this framework as a starting point and define a combined update and query rule using a recurrent neural network parameterized by θ such that

$$\mathbf{h}_t, \mathbf{x}_t = RNN_\theta(\mathbf{h}_{t-1}, \mathbf{x}_{t-1}, y_{t-1}) \qquad (6)$$

$$y_t \sim p(y|x_t) \qquad (7)$$

4.2.1 Loss Functions

Now they need loss function to learn the parameters. One of the Loss functions they use is defined as summed loss:

$$L_{sum}(\theta) = \mathbb{E}_{f,y_{1:T-1}}\left[\sum_{t=1}^{T} f(x_t)\right] \qquad (8)$$

A key reason to, they say, prefer L_{sum} over the final loss is that the amount of information conveyed final Loss is temporally very sparse. By instead utilizing a sum of losses to train the optimizer they are able to provide information from every step along this trajectory. Although at test time the optimizer typically only has access to the observation y_t, at training time the true loss can be used.

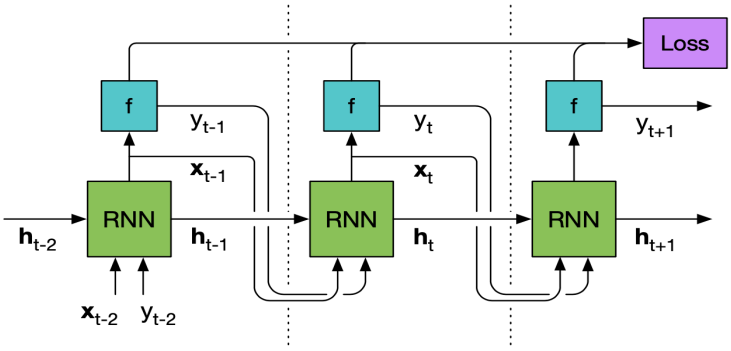

Figure 3: Computational graph of the learned black-box optimizer unrolled over multiple steps.[2]

We can encourage exploration in the space of optimizers by encoding an exploratory force directly into the meta learning loss function. But this might not give the best performance as the actual process of optimizing the above loss can be difficult due to the fact that nothing explicitly encourages the optimizer itself to explore.

Encoding an exploratory force directly into the meta learning loss function can encourage exploration in the space of optimizers. For example;

$$L_{EI}(\theta) = -\mathbb{E}_{f,y_{1:T-1}}\left[\sum_{t=1}^{T} EI(\mathbf{x}_t|y_{1:T-1})\right] \qquad (9)$$

where EI(·) is the expected posterior improvement of querying $\mathbf{x}_t$ given observations up to time t. This can encourage exploration by giving an explicit bonus to the optimizer rather than just implicitly doing so by means of function evaluations.

They have also used the observed improvement (OI)

$$L_{OI}(\theta) = \mathbb{E}_{f,y_{1:T-1}}\left[\sum_{t=1}^{T} \min\left\{f(x_t) - \min_{i<t}(f(x_i)), 0\right\}\right] \qquad (10)$$

4.2.2 Gaussian Process

To this point no assumptions has been made about the distribution of training functions $p(f)$. They propose the use of GPs as a suitable training distribution. A Gaussian process is a probability distribution over possible functions that fit a set of points. It begins with a prior distribution and updates this as data points are observed, producing the posterior distribution over functions. Covariance matrix, along with a mean function to output the expected value of $f(x)$ defines a Gaussian Process.

According to [7] Here's how Kevin Murphy explains it in the excellent textbook Machine Learning: A Probabilistic Perspective:

A GP defines a prior over functions, which can be converted into a posterior over functions once we have seen some data. Although it might seem difficult to represent a distribution over a function, it turns out that we only need to be able to define a distribution over the function's values at a finite, but arbitrary, set of points, say $x_1, \ldots, x_N$. A GP assumes that $p(f(x_1), \ldots, f(x_N))$is jointly Gaussian, with some mean $\mu(x)$ and covariance $\sum(x)$ given by $\sum_{ij} = k(x_i, x_j)$, where k is a positive definite kernel function. The key idea is that if x_i and x_j are deemed by the kernel to be similar, then we expect the output of the function at those points to be similar, too.

Under the GP prior, the joint distribution of function values at any finite set of query points follows a multivariate Gaussian distribution [8], and we generate a realization of the training function incrementally at the query points using the chain rule with a total time complexity of $O(T^3)$ for every function sample. The use of functions sampled from a GP prior also provides functions whose

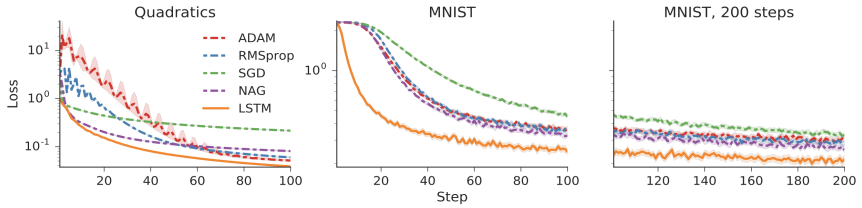

Figure 4: Comparisons between learned and hand-crafted optimizers performance. Learned optimizers are shown with solid lines and hand-crafted optimizers are shown with dashed lines. Units for the y axis in the MNIST plots are logits. **Left:** Performance of different optimizers on randomly sampled 10-dimensional quadratic functions. **Center:** the LSTM optimizer outperforms standard methods training the base network on MNIST. **Right:** Learning curves for steps 100-200 by an optimizer trained to optimize for 100 steps (continuation of center plot).[1]

gradients can be easily evaluated at training time as noted above. The major downside of search strategies which are based on GP inference is their cubic complexity.

4.2.3 Parallel Function Evaluation

Parallel Function Evaluation refers to training several networks in parallel. How this is done is explained in [2] as follows: Suppose there are N workers, and that the process of proposing candidates for function evaluation is much faster than evaluating the functions. They augment RNN optimizer's input with a binary variable o_t as follows:

$$h_t, x_t = RNN_\theta(h_{t-1}, o_{t-1}, \tilde{x}_{t-1}, \tilde{y}_{t-1}) \qquad (11)$$

For the first $t \leq N$ steps, they set $o_{t-1} = 0$, arbitrarily set the inputs to dummy values $\tilde{x}_{t-1} = 0$ and $\tilde{y}_{t-1} = 0$, and generate N parallel queries $x_{1:N}$. As soon as a worker finishes evaluating a query, the query and its evaluation are fed back to the network by setting $o_{t-1} = 1$, resulting in a new query x_t. The architecture allows for the number of workers to vary.

5 Results

5.1 RNN as gradient descent

In [1] all experiments the trained optimizers use two-layer LSTMs with 20 hidden units in each layer. The minimization is performed using ADAM with a learning rate chosen by random search. Early stopping is used when training the optimizer in order to avoid overfitting the optimizer. After each epoch (some fixed number of learning steps) they freeze the optimizer parameters and evaluate its performance. The best optimizer (according to the final validation loss) and report its average performance on a number of freshly sampled test problems. Comparison in done between trained optimizers with standard optimizers used in Deep Learning: SGD, RMSprop, ADAM, and Nesterov's accelerated gradient (NAG). For each of these optimizer and each problem tuned the learning rate, and report results with the rate that gives the best final error for each problem. Initial values of all optimizee parameters were sampled from an IID Gaussian distribution.

5.1.1 Quadratic

In this experiment they have trained an optimizer on a simple class of synthetic 10-dimensional quadratic functions. In particular minimizing functions of the form

$$f(\theta) = \|W\theta - y\|_2^2. \qquad (12)$$

for different 10x10 matrices W and 10-dimensional vectors y whose elements are drawn from an IID Gaussian distribution. Optimizers were trained by optimizing random functions from this family and tested on newly sampled functions from the same distribution. Each function was optimized for 100 steps and the trained optimizers were unrolled for 20 steps. They have not used any preprocessing,

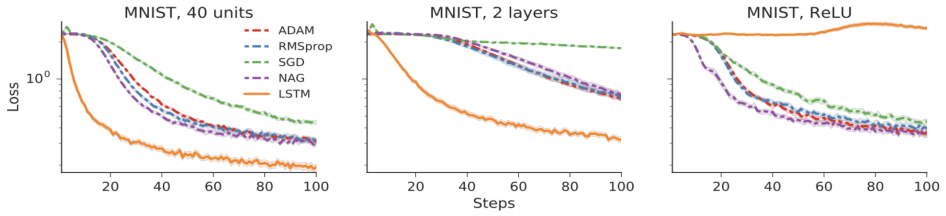

Figure 5: Comparisons between learned and hand-crafted optimizers performance. Units for the y axis are logits. Here we can clearly see the generalization capability of the learned optimizer

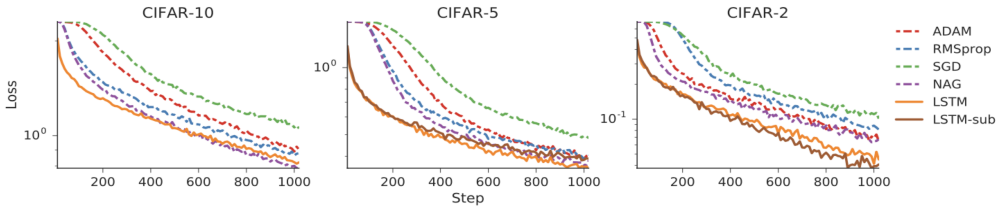

Figure 6: Optimization performance on the CIFAR-10 dataset and subsets

nor post processing. Learning curves for different optimizers, averaged over many functions, is shown in figure 1. Each curve corresponds to the average performance of one optimization algorithm on many test functions; the solid curve shows the learned optimizer performance and dashed curves show the performance of the standard baseline optimizers. It is clear the learned optimizers based in LSTM substantially outperform the baselines.

5.1.2 Training A Small Neural Network On MNIST

This experiment is performed to optimize a small neural network on MNIST dataset. In this experiment they shows this trainable optimizer based on LSTM can learn to optimize a small neural network on MNIST, and also shows its performance on novel architecture. It generalize well to functions beyond those it was trained on. to show this they train the optimizer to optimize a base network and explore a series of modifications to the network architecture and training procedure at test time. The base network is an MLP with one hidden layer of 20 units using a sigmoid activation function. The only source of variability between different runs is the initial value θ_0 and randomness in minibatch selection. Each optimization was run for 100 steps and the trained optimizers were unrolled for 20 steps. Learning curves for the base network using different optimizers are displayed. In this experiment NAG, ADAM, and RMSprop exhibit roughly equivalent performance the LSTM optimizer outperforms them by a significant margin.

The right plot in Figure 4 compares the performance of the LSTM optimizer if it is allowed to run for 200 steps, although it has been trained to optimize for 100 steps. In this comparison we re-used the LSTM optimizer from the previous experiment, and here we see that the LSTM optimizer continues to outperform the baseline optimizers on this task.

Figure 5 shows three examples of applying the LSTM optimizer to train networks with different architectures than the base network on which it was trained. The modifications are (from left to right) (1) an MLP with 40 hidden units instead of 20, (2) a network with two hidden layers instead of one, and (3) a network using ReLU activations instead of sigmoid. In the first two cases the LSTM optimizer generalizes well, and continues to outperform the hand-designed baselines despite operating outside of its training regime. However, changing the activation function to ReLU makes the dynamics of the learning procedure sufficiently different that the learned optimizer is no longer able to generalize.

Figure 7: Optimization result for NeuralArt

5.1.3 Training a Neural network on CIFAR-10

Optimization performance on the CIFAR-10 dataset and subsets. Shown in Figure 6 is the LSTM optimizer versus various baselines trained on CIFAR-10 and tested on a held-out test set. The two plots on the right are the performance of these optimizers on subsets of the CIFAR labels. The additional optimizer LSTM-sub has been trained only on the heldout labels and is hence transferring to a completely novel dataset. ie. the CIFAR-2 dataset only contains data corresponding to 2 of the 10 labels. Additionally we include an optimizer LSTM-sub which was only trained on the held-out labels. The coordinatewise network decomposition introduced in Section 4.1.2 and used in the previous experiment was not sufficient for the model architecture introduced in this section due to the differences between the fully connected and convolutional layers. Instead they modify optimizer in such a way that the previous LSTM optimizer still utilizes a coordinatewise decomposition with shared weights and individual hidden states, however LSTM weights are now shared only between parameters of the same type (i.e. fully-connected vs. convolutional). In all these examples we can see that the LSTM optimizer learns much more quickly than the baseline optimizers, with significant boosts in performance for the CIFAR-5 and especially CIFAR-2 datsets. We also see that the optimizers trained only on a disjoint subset of the data is hardly effected by this difference and transfers well to the additional dataset.

5.1.4 Neural Art

In Neural Art we have a Content Image and a Style Image and we want to generate a new smooth image which combining these two. Examples of images styled using the LSTM optimizer.In Figure 7 Each triple consists of the content image (left), style (right) and image generated by the LSTM optimizer (center). The Left side shows the result of applying the training style at the training resolution to a test image. The right side shows the result of applying a new style to a test image at double the resolution on which the optimizer was trained.

5.2 RNN as BlackBox Optimization

Several experiments are presented to show the breadth of generalization that is achieved by the learned algorithms. The algorithms are trained to optimize very simple functions—samples from a GP with a fixed length scale. The learned algorithms are shown to be able to generalize from these simple objective functions to a wide variety of other test functions that were not seen during training. Experiments are done using two different RNN architectures: LSTMs and DNCs. However, they show that the DNCs perform slightly (but not significantly) better.

Each RNN optimizer is trained with trajectories of T steps, and parameters are updated using BPTT with Adam. They use a curriculum to increase the length of trajectories gradually from T = 10 to 100. This process is repeated for each of the loss functions discussed in Section 4.

Hyper-parameters for the RNN optimization algorithm (such as learning rate, number of hidden units, and memory size for the DNC models) are found through grid search during training. When ready to be used as an optimizer, the RNN requires neither tuning of hyper-parameters nor hand engineering. It is made fully automatic.

In the following experiments, DNC sum refers to the DNC network trained using the summed loss L_{sum}, DNC OI to the network trained using the loss L_{OI}, and DNC EI to the network trained with the loss L_{EI}. Comparison is shown for this learning to learn approach with popular state-of-

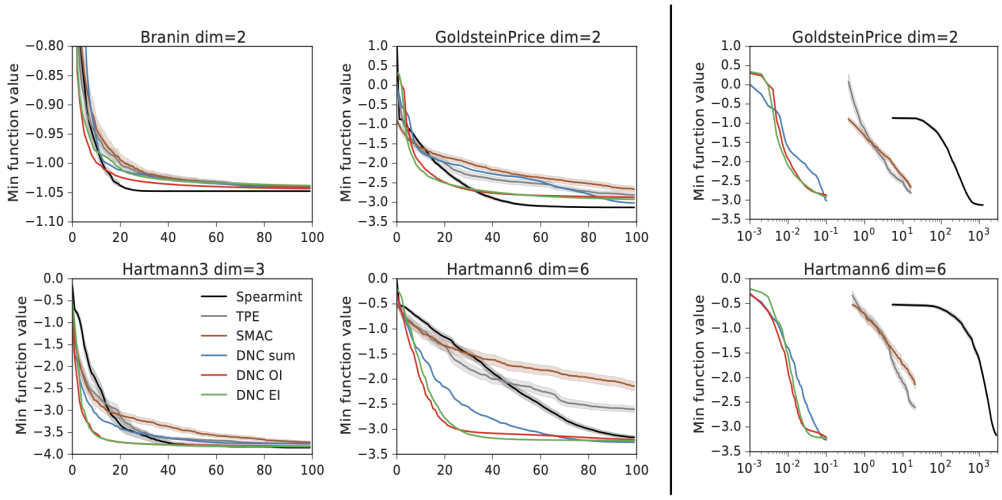

Figure 8: Average minimum observed function value, with 95% confidence intervals, as a function of search steps on functions sampled from the training GP distribution. Left four figures: Comparing DNC with different reward functions against Spearmint with fixed and estimated GP hyper-parameters, TPE and SMAC. Right bottom: Comparing different DNCs and LSTMs. As the dimension of the search space increases, the DNC's performance improves relative to the baselines.[2]

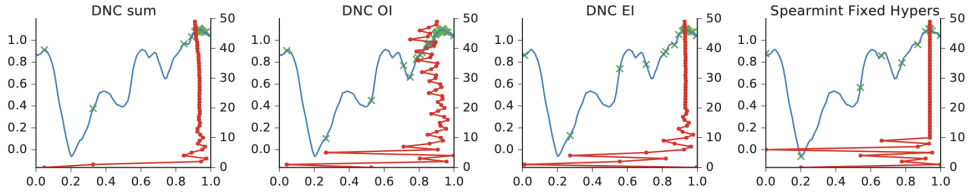

Figure 9: How different methods trade-off exploration and exploitation in a one-dimensional example. Blue: Unknown function being optimized. Green crosses: Function values at query points. Red trajectory: Query points over 50 steps.[2]

the-art Bayesian optimization packages, including Spearmint with automatic inference of the GP hyperparameters.

5.2.1 Performance on Functions Sampled from the Training Distribution

They show performance on functions sampled from the training distribution. Notice, however, that these functions are never observed during training.Figure 8 shows the best observed function values as a function of search step t, averaged over 10, 000 sampled functions for RNN models and 100 sampled functions for other models (RNNs have more because they are very fast optimizers). For Spearmint, they consider two different scenarios for hyper-parameter setting . One of which is default setting with a prior distribution that estimates the GP hyperparameters by Monte Carlo and another one is the setting with the same hyper-parameters as those used in training. For the second setting, Spearmint knows the ground truth and thus provides a very competitive baseline. As expected Spearmint with a fixed prior proves to be one of the best models under most settings.

When the input dimension is 6 or higher, however, neural network models start to outperform Spearmint. It might be because in higher dimensional spaces, the RNN optimizer learns to be more exploitative given the fixed number of iterations. Among all RNNs, those trained with expected/observed improvement perform better than those trained with direct function observations.

Figure 9 shows the query trajectories. $\mathbf{x}_t, t = 1, ..., 100$, for different black-box optimizers in a one-dimensional example. All of the optimizers explore initially, and later settle in one mode and

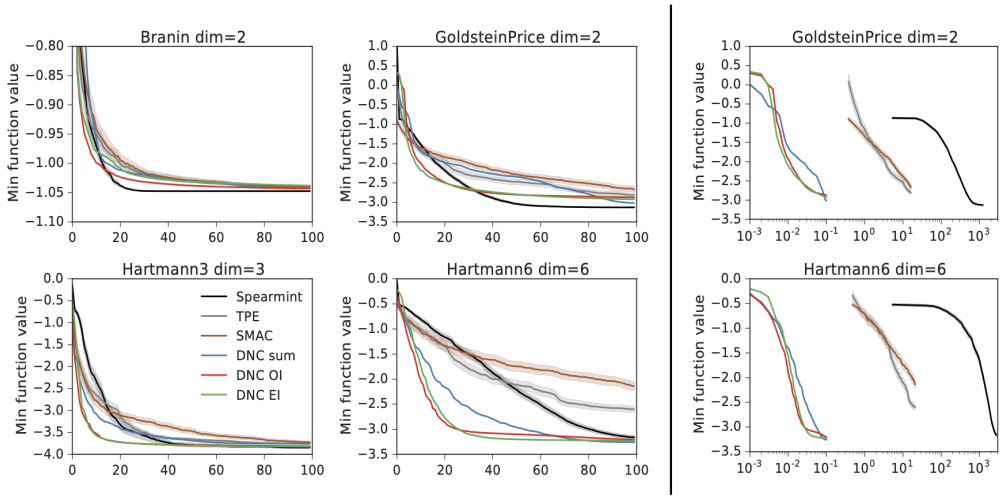

Figure 10: Average minimum observed function value, with 95% confidence intervals, as a function of search steps on functions sampled from the training GP distribution. Left four figures: Comparing DNC with different reward functions against Spearmint with fixed and estimated GP hyper-parameters, TPE and SMAC. Right bottom: Comparing different DNCs and LSTMs. As the dimension of the search space increases, the DNC's performance improves relative to the baselines.[2]

search more locally. The DNCs trained with EI behave most similarly to Spearmint. DNC with direct function observations $(DNCsum)$ tends to explore less than the other optimizers and often misses the global optimum, while the DNCs trained with the observed improvement (OI) keep exploring even in later stages.

5.2.2 Transfer to Global Optimization Benchmarks

They compare the algorithms on four standard benchmark functions for black-box optimization with dimensions ranging from 2 to 6. To obtain a more robust evaluation of the performance of each model, multiple instances for each benchmark function is generated by applying a random translation (-0.1-0.1), scaling (0.9-1.1), flipping, and dimension permutation in the input domain. The left hand side of Figure 5 shows the minimum observed function values achieved by the learned DNC optimizers, and contrasts these against the ones attained by Spearmint, TPE and SMAC. All methods appear to have similar performance with Spearming doing slightly better in low dimensions. As the dimension increases, we see that the DNC optimizers converge at at a much faster rate within the horizon of T = 100 steps. We also observe that DNC OI and DNC EI both outperform DNC with direct obsevations of the loss (DNC sum). It is encouraging that the curves for DNC OI and DNC EI are so close. While DNC EI is distilling a popular acquisition function from the EI literature, the DNC OI variant is much easier to train as it never requires the GP computations nec- essary to construct the EI acquisition function.

The right hand side of Figure 10 shows that the neural network optimizers run about 104 times faster than Spearmint and 102 times faster than TPE and SMAC with the DNC architecture. There is an additional 5 times speedup when using the LSTM architecture. The negligible runtime of our optimizers suggests new areas of application for global optimization methods that require both high sample efficiency and real-time performance.

5.2.3 Transfer to a Simple Control Problem

In this they show an application to a simple reinforcement learning[2]. In this problem we simulate a physical system consisting of a number of repellers which affect the fall of particles through a 2D-space. The goal is to direct the path of the particles through high reward regions of the state space and maximize the accumulated discounted reward. The four dimensional state-space in this problem consists of a particle's position and velocity. The path of the particles can be controlled by

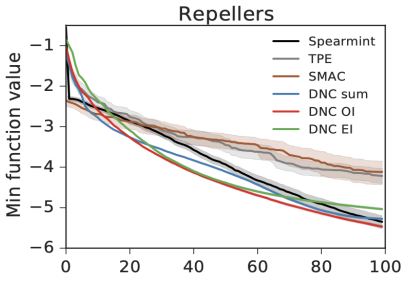

Figure 11: The results of each method on optimizing the controller by direct policy search. Here, the learned DNC OI optimizer appears to have an edge over the other techniques.

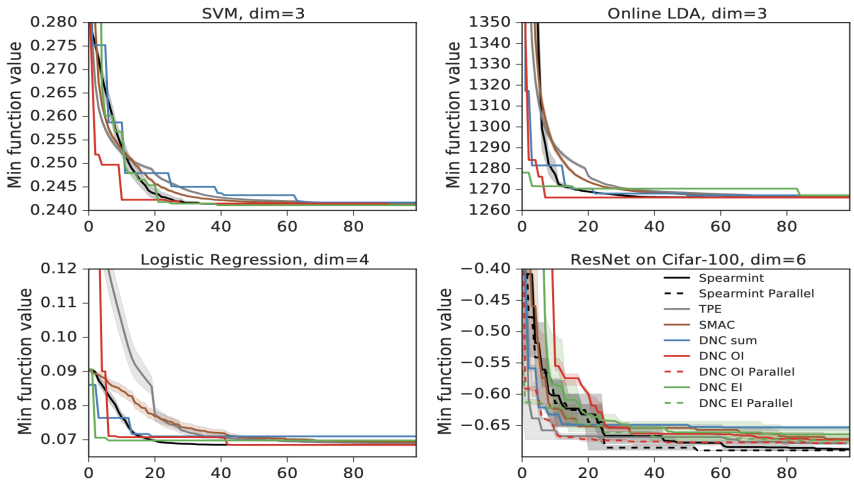

Figure 12: Average test loss, with 95% confidence intervals, for the SVM, online LDA, and logistic regression hyper-parameter tuning benchmarks. The bottom-right plot shows the performance of all methods on the problem of tuning a residual network, demonstrating that the learned DNC optimizers are close in performance to the engineered optimizers, and that the faster parallel versions work comparably well.

the placement of repellers which push the particles directly away with a force inversely proportional to their distance from the particle. At each time step the particle's position and velocity are updated using simple deterministic physical forward simulation. The control policy for this problem consists of 3 learned parameters for each repeller: 2d location and the strength of the repeller. Experiments are done considering with 2 repellers, i.e. 6 parameters. We apply the same perturbation as in the previous subsection to study the average performance. The loss (minimal negative reward) of all models are plotted in Figure 11. Neural network models outperform all the other competitors in this problem.

5.2.4 Transfer to ML Hyper-parameter Tuning

Lastly, they consider hyper-parameter tuning for machine learning problems. They include the three standard benchmarks in the HPOLib package[2]: SVM, online LDA, and logistic regression with 3, 3, and 4 hyper-parameters respectively with the problem of training a 6-hyper-parameter residual network for classification on the CIFAR-100 dataset. For the first three problems, the objective functions have already been pre-computed on a grid of hyper-parameter values, and therefore evaluation with different random seeds (100 for Spearmint, 1000 for TPE and SMAC) is cheap. For the last experiment, however, it takes at least 16 GPU hours to evaluate one hyper-parameter setting. For this reason, they use the parallel proposal idea introduced in Section 2.3, with 5 parallel proposal

mechanisms. This approach is about five times more efficient. For the first three tasks, they run the model once because the setup is deterministic. For the residual network task, there is some random variation so they consider three runs per method. The results are shown in Figure 12. The plots report the negative accuracy against number of function evaluations up to a horizon of T = 100. The neural network models especially when trained with observed improvement show competitive performance against the engineered solutions. In the ResNet experiment, they also compare our sequential DNC optimizers with the parallel versions with 5 workers. In this experiment we find that the learned and engineered parallel optimizers perform as well if not slightly better than the sequential ones. These minor differences arise from random variation.

6 Conclusions and future work

Optimizer RNN They have shown how to cast the design of optimization algorithms as a learning problem using recurrent neural networks, which enables us to train optimizers that are specialized to particular classes of functions. Experiments have confirmed that learned neural optimizers compare favorably against state-of-the-art optimization methods used in deep learning. We witnessed a remarkable degree of transfer, with for example the LSTM optimizer trained on 12,288 parameter neural art tasks being able to generalize to tasks with 49,152 parameters, different styles, and different content images all at the same time. We observed similar impressive results when transferring to different architectures in the MNIST task. The results on the CIFAR image labeling task show that the LSTM optimizers outperform hand- engineered optimizers when transferring to datasets drawn from the same data distribution.

As for this we can clearly see that the proposed method performs better in most scenarios than the hand-engineered one. But As we use neural network in this we will have to take into account the training time associated as they can increase. Also this makes training computationally expensive.

RNN for Black-Box The experiments have shown that up to the training horizon the learned RNN optimizers are able to match the performance of heavily engineered Bayesian optimization solutions, including Spearmint, SMAC and TPE. The trained RNNs rely on neither heuristics nor hyper-parameters when being deployed as black-box optimizers. The optimizers trained on synthetic functions were able to transfer successfully to a very wide class of black-box functions as seen.

The experiments have also shown that the RNNs are massively faster than other Bayesian optimization methods. However, the current RNN optimizers also have some shortcomings. Training for very long horizons is difficult. Authors believe curriculum learning should be investigated as a way of overcoming this difficulty. In addition, a new model has to be trained for every input dimension with the current network architecture. While training optimizers for every dimension is not prohibitive in low dimensions, future works should extend the RNN structure to allow a variable input dimension. A promising solution is to serialize the input vectors along the search steps.

References

[1] Andrychowicz, Marcin, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. "Learning to learn by gradient descent by gradient descent." *Advances in neural information processing systems 29* (2016).

[2] Chen, Yutian, Matthew W. Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P. Lillicrap, Matt Botvinick, and Nando Freitas. "Learning to learn without gradient descent by gradient descent." *In International Conference on Machine Learning*, pp. 748-756. PMLR, 2017.

[3] Schmidt, Robin M. "Recurrent neural networks (rnns): A gentle introduction and overview." *arXiv preprint arXiv:1912.05911* (2019).

[4] Staudemeyer, Ralf C., and Eric Rothstein Morris. "Understanding LSTM–a tutorial into long short-term memory recurrent neural networks." *arXiv preprint arXiv:1909.09586* (2019).

[5] Graves, Alex, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo et al. "Hybrid computing using a neural network with dynamic external memory." *Nature* 538, no. 7626 (2016): 471-476.

[6] Wei Wang, "Bayesian Optimization Concept Explained in Layman Terms," Towards Data Science, March 18, 2020, https://towardsdatascience.com/bayesian-optimization-concept-explained-in-layman-terms-1d2bcdeaf12f

[7] Katherine Bailey, "Gaussian Processes for Dummies," Disqus, August 9, 2016, https://katbailey.github.io/post/gaussian-processes-for-dummies/#disqus_thread.

[8] Rasmussen, Carl Edward and Williams, Christopher K. I.. *Gaussian processes for machine learning.*. : MIT Press, 2006.