# Black Box Optimization using Recurrent Neural Networks

Pattarawat Chormai, Felix Sattler, Raphael Holca-Lammare

August 1, 2017

**Abstract**

We extend on the work of Chen et al., who introduced a new, learning-based, approach to global Black-Box optimization [2]. They train a LSTM model [9], on functions sampled from a Gaussian process, to output an optimal sequence of sample points, i.e. a sequence of points that minimize those training functions. We verify the claims made by the authors, confirming that such a trained model is able to generalize to a wide variety of synthetic Black-Box functions, as well as to the real world problems of airfoil optimization and hyperparameter tuning of a SVM classifier. We show that the method performs comparably to state-of-the-art Black-Box optimization algorithms on these benchmarks, while outperforming them by a wide margin w.r.t. computation time. We thoroughly investigate the effects of different loss functions and training distributions on the methods performance and examine ways to improve it in the presence of prior knowledge.

## 1 Introduction

In industry and science we frequently encounter problems of the type

$$\text{"Find } x^* \in \Theta, \text{ s.t. } f(x^*) \leq f(x) \text{ for all } x \in \Theta\text{"} \tag{1}$$

for some objective function $f : \Theta \subset \mathbb{R}^d \to \mathbb{R}$. In many cases, little to no information is available about properties of $f$. In particular, no gradient information can be used to guide the optimization process in these cases. The optimization problem (1) is then referred to as "Black-Box-Optimization". Evaluation of the objective function $f$ is often costly and thus usually the practical goal is to find as small of a function value as possible in a fixed amount of steps $n$.

$$\text{"Find a sequence } x_1, .., x_n \in \Theta, \text{ s.t.} \min(f(x_1), .., f(x_n)) \to \min\text{"} \tag{2}$$

There exist already many approaches to solving (2). Methods based on simulated annealing [13] perform a guided local random search through the space of possible solutions. Surrogate based methods [5] build a model of the objective function based on previously evaluated sample points. They then decide on where to sample next based on this internal surrogate model. A special kind of surrogate based optimization is Bayesian optimization [18][19], in which the surrogate model is given by a posterior distribution over some function space. Knowledge about the objective function can be included in these methods by the choice of a suitable prior. An exhaustive comparison of black box optimization algorithms can be found in [17].

In their 2017 paper "Learning to learn without gradient descent, by gradient descent" [2] Yutian Chen et al. present a completely novel, learning based, approach to Black-Box optimization that makes use of Recurrent Neural Networks (RNNs). RNNs have achieved stunning results in many sequence modeling tasks. They revolutionized the fields of speech recognition [6], machine translation [1] and image annotation [20], beating the former state-of-the-art in many cases by wide margins. As indicated by (2), we can interpret Black-Box optimization as a task of finding an optimal sequence, thus making it possible to harness the power of RNNs.

Our contribution in this paper is as follows: We re-implement the method described by Chen et al. and reproduce some of their experiments, confirming the remarkable performance of their algorithm with respect to computation time and minimum function value found. We extensively investigate the effects of different loss functions and training data distributions on the performance of the algorithm and explore the benefits of incorporating prior knowledge into the optimizer by augmenting the training data distribution to the objective function. We explore limitations of the method and report results on the real world problems of airfoil-optimization [4] and hyperparameter tuning of an SVM classifier [3]. For the sake of simplicity we will refer to our re-implementation of Chen et al.'s algorithm in the following as "our method".

## 2 The Model

We will model our optimizer as a Long-Short-Term-Memory RNN (short LSTM). LSTMs were first introduced by Hochreiter & Schmidhuber in [9]. They have become a popular RNN model in recent years, because they do not suffer from what is called the "vanishing gradient problem" [8]. This makes it a lot easier to train LSTMs for long sequences compared to vanilla RNNs. Starting from some fixed initial point $x_0 = x_{\text{start}}$ and empty hidden state $h_0 = 0 \in \mathbb{R}^{n_h}$, we define the following recursive update-rule based on a LSTM parametrized by $\theta$:

$$\begin{aligned} x_n, h_n &= \text{LSTM}_\theta(x_{n-1}, y_{n-1}, h_{n-1}) \\ y_n &= f(x_n) \end{aligned} \tag{3}$$

The LSTM accumulates information about previous sample points $x_1, .., x_{n-1}$ and their function values $f(x_1), .., f(x_{n-1})$ in a hidden state $h$. Based on this hidden state it decides on where to sample next. The hidden state can thus be interpreted as the models internal representation of the objective function. The parameters of the model, $\theta$, are then trained on a suitable distribution of differentiable functions $\psi(f)$ to minimize the average observed minimal function value

$$\min_\theta \int \min_{k=1,..,n} f(x_k) d\psi(f). \tag{4}$$

This is practically done by performing stochastic gradient descent on $\theta$ w.r.t. some suitable loss function $L$

$$\Delta\theta = -\frac{1}{\text{batchsize}} \sum_{i=1}^{\text{batchsize}} \nabla_\theta L(f^{(i)}, x_0, x_1^{(i)}, .., x_n^{(i)}), \quad f^{(i)} \sim \psi(f). \tag{5}$$

Note that although the model can only be trained on differentiable functions, it can be evaluated on any kind of Black-Box function. The model is sketched at two different levels of abstraction in Figure 1.

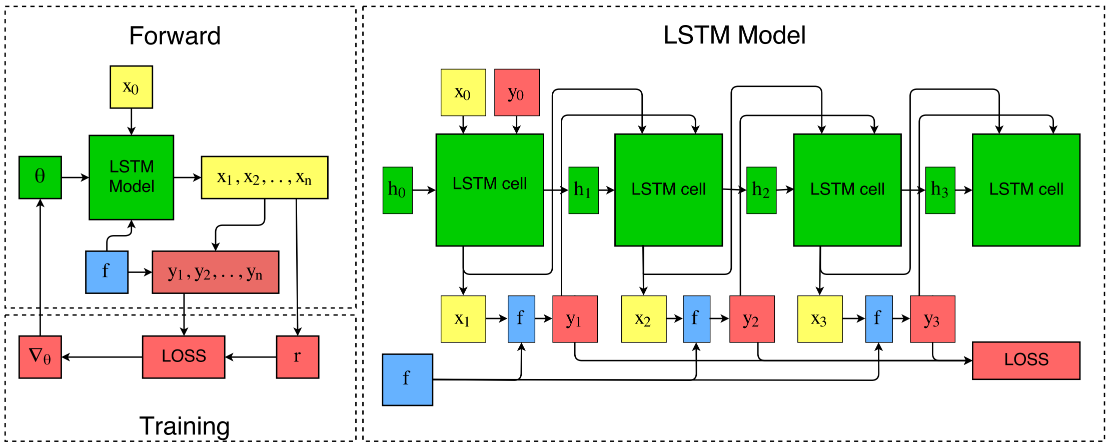

Figure 1: Computational graph of the model at two different levels of abstraction. Left: For every function $f$ and starting point $x_0$ the model returns a sequence of sample points $x_1, .., x_n$ based on it's internal parameters $\theta$. During training these parameters are updated so as to reduce the average loss over some distribution of training functions. Right: The LSTM Model unrolled over several steps.

### 2.1 Loss-Functions

The model learns to trade-off exploration and exploitation based on the loss function that is used during training. We experimented with a couple of different loss functions. The most straight

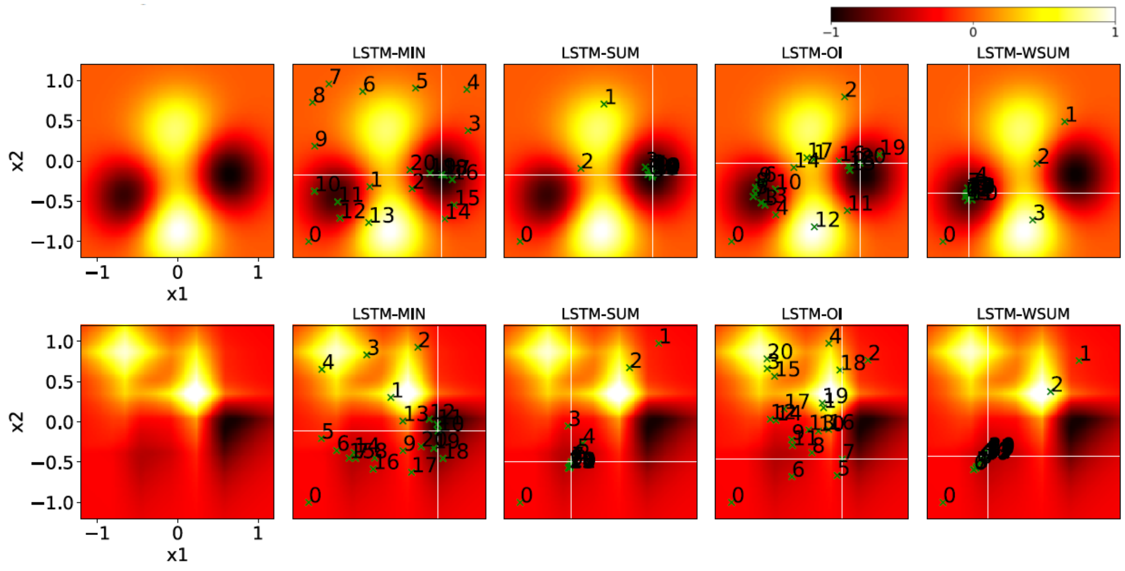

Figure 2: Different sampling behaviors for different loss functions on functions sampled from a Gaussian process with RBF kernel (top row) and Matern32 kernel (bottom row). Loss functions from left to right: MIN, SUM, OI, WSUM.

forward approach is to train directly w.r.t. to goal (4) and thus use

$$L_{\text{MIN}}(f, x_0, ..x_n) = \min_{k=1,..,n} f(x_k).$$

Chen et al. argue that the information provided by this loss function is sparse, since only the sample point with the lowest function value directly contributes to the loss. They deduce from that, that the gradient signal coming from this loss ought to be non-informative and unsuited for training. Our experiments contradict this line of thought as we will see in the next section. Chen et al. instead resort to the observed improvement loss, given by

$$L_{\text{OI}}(f, x_0, .., x_n) = \sum_{k=1}^{n} \min(0, f(x_k) - \min_{i<k} f(x_i)).$$

where the term $\min_{i<k} f(x_i)$ is considered as a constant during optimization. They argue that this loss should produce richer gradients because more sample points contribute. Both MIN and OI loss encourage exploratory sampling behavior, because sample points with high function values do not contribute to the loss.

Chen et al. also experiment with the summed loss

$$L_{\text{SUM}}(f, x_0, .., x_n) = \sum_{k=1}^{n} f(x_k)$$

which corresponds to the cumulative regret over all time steps. Optimizing for the summed loss yields a very greedy search strategy as exploration to areas with high function values is punished. We investigated the weighted sum loss as a way to overcome this problem. The idea is to assign less weight to early sample points to encourage exploration.

$$L_{\text{WSUM}}(f, x_0, .., x_n) = \sum_{k=1}^{n} w_k f(x_k)$$

There are many ways to choose the values for $w_k$. Some of them include exponential decay $w_k = e^{-(n-k)}$, linear decay $w_k = \frac{k}{n}$ or complete suppression $w_k = \delta_{k \geq k_0}$. We experimented mainly with exponential decay. To our surprise we found that the weighted sum loss suppresses the greedy behavior only marginally. Figure 2 shows the characteristic sampling behavior we get from the different loss functions.

## 2.2 Training Data

Theoretically, it is possible to train the model on any kind of function distribution $\psi(f)$, as long as every function sampled from the distribution is differentiable. But, as for every learning algorithm, the choice of the training distribution will greatly determine the behavior on test data. If prior knowledge about the objective function is available this can (and should) be incorporated into the training distribution as it will likely improve the performance of the optimizer, much like an image classifier should be trained on images similar to those it is supposed to classify at test time. We will provide further evidence for this in the next section. In the most general case however, when no prior information is available, one would expect the model to generalize the better, the broader distribution of training data. We therefore train our model on Gaussian process [16] data $\psi(f) = g_{X,Y}$. For a fixed kernel $k$, a Gaussian process function in $d$ dimensions, is uniquely determined by a set of supporting points $X = \{x_1, .., x_m\} \in \mathbb{R}^{m,d}$ and corresponding function values $Y = \{y_1, .., y_m\} \in \mathbb{R}^{m,1}$. It is given by

$$g_{X,Y}(x) = \mathbb{E}(\mathcal{GP}_{X,Y}(x)) = (K_{X,X}^{-1}Y)^T K_X(x) \tag{6}$$

with kernel matrices

$$K_{X,X} = [k(x_i, x_j)]_{i,j=1,..,m} \in \mathbb{R}^{m,m}$$

and

$$K_X(x) = [k(x_i, x)]_{i=1,..,m} \in \mathbb{R}^{m,1}$$

Every such GP function passes through all of it's supporting points: $g_{X,Y}(x_i) = y_i, \quad i = 1, .., m$. In between supporting points, the behavior of the function is determined by the kernel. We trained with two different kernels: The RBF-kernel

$$k_{\text{rbf}}(x_1, x_2) = \exp(-\frac{\|x_1 - x_2\|^2}{2l^2})$$

produces functions that are infinitely differentiable, $g_{X,Y}^{\text{rbf}} \in \mathcal{C}^\infty$. Whereas the Matern32-kernel

$$k_{\text{matern32}}(x_1, x_2) = (1 + \frac{\gamma\sqrt{3}}{l}\|x_1 - x_2\|^2)\exp(-\frac{\gamma\sqrt{3}}{l}\|x_1 - x_2\|^2)$$

produces functions that are continuous and a.e. differentiable, $g_{X,Y}^{\text{mat}} \in \mathcal{C}$. Since computing the kernel matrix $K_{X,X}$ and its inverse can be quite time-consuming, it is advisable to pre-compute it before starting the training process.

## 3 Training Practicalities

### 3.1 Regularization

Black-Box optimization is usually performed on a compact region. Since every function on a compact domain can be rescaled arbitrarily, it is sufficient to only optimize on the centered hypercube of side-length 2, $\Theta = [-1, 1]^d$. This also conveniently imposes an output normalization onto our model, which is beneficial to training. Since the output of the LSTM is unbounded, it is necessary to force it externally to only output values that lie in the valid region. We first tried to resolve this issue by adding an additional tanh activation to the output. The problem with this approach is that it causes an undesirable bias towards sampling in the center of the valid region, because sampling close to the border requires very high pre-activation values. Therefore we ultimately decided to instead go with a regularization approach. We added to the loss a high multiple of the sample points $l_\infty$-distance to the valid region given by

$$r(x_0, .., x_n) = c\sum_{i=1}^{n}(\max(1, \|x_i\|_\infty) - 1), \tag{7}$$

where $c$ is a big constant (e.g. $c = 1000$). This leaves points inside the valid region untouched and only punishes the model if it produces points that are invalid. Using this approach, we were able to prevent the model from ever sampling in invalid regions, without any negative side-effects on the performance.

## 3.2 Normalizing the Training Data

Although we were able to get the model to train with functions directly sampled from $g_{X,Y}$, we observed that it is possible to achieve much better results if we normalize these functions first. Normalizing the training data, also known as feature scaling, is a very common preprocessing step for many machine learning algorithms, see for example [7], [10]. We computed approximations to upper and lower bounds of all functions via Monte-Carlo sampling

$$y_{min} \approx \min_{x_1,..,x_j \sim \mathcal{U}([-1,1]^d)} f(x), \quad y_{max} \approx \max_{x_1,..,x_j \sim \mathcal{U}([-1,1]^d)} f(x) \tag{8}$$

and then used these values to normalize the function to

$$\tilde{g}_{X,Y}(x) = 2\frac{g_{X,Y}(x) - y_{min}}{y_{max} - y_{min}} - 1, \tag{9}$$

where now $\tilde{g}_{X,Y} : [-1,1]^d \to [-1,1]$. In practice, such upper an lower bounds are often known even for Black-Box functions. For example, in the case of hyperparameter optimization for an image classifier we might have no idea what influence certain hyperparameters have on the classification error, but we know for sure that this error will always be bound between 0 and 1. We will restrict our analysis in the following to such "normalizable" functions. It remains an open question for further research how the method can be adapted to general Black-Box functions, for which no such bounds are available.

The whole training procedure is described in Algorithm (1).

**Algorithm 1:** LSTM Training

```
for number of training iterations do
    • Sample a mini-batch of functions $f^{(i)} = g_{X_i,Y_i}$, $X_i \sim \mathcal{U}([-1,1]^{(m,d)})$, $Y_i \sim \mathcal{U}([-1,1]^{(m,1)})$,
      $i \in \{1,..,\text{batchsize}\}$ according to (6)
    • Approximate upper and lower bounds to all functions and use these to normalize the functions
      according to (8),(9)
    • Compute, for every function $f^{(i)}$, a candidate sampling sequence $x_1^{(i)},..x_n^{(i)}$ (forward pass) and update
      the parameters $\theta$ of the LSTM model using stochastic gradient descent (backward pass) according to
          $\Delta\theta = -\frac{1}{\text{batchsize}} \sum_{i=1}^{\text{batchsize}} \nabla_\theta L(f^{(i)}, x_0, x_1^{(i)}, ..x_n^{(i)}) + \nabla_\theta r(x_0, x_1^{(i)}, ..x_n^{(i)})$
end
```

# 4 Experimental Results

We trained our model on normalized functions sampled from a Gaussian process with either a rbf or a matern32 kernel in different dimensions. We initialized the LSTM with a low forget bias and used gradient clipping to avoid exploding gradients, following the recommendations in [15]. We used the momentum based Adam Optimizer [12] and an exponentially decaying learning rate during training. There is no risk of overfitting with our method, because the amount of available training data is infinite. That said, we discovered that the amount of training functions needed to achieve a good generalization increases drastically with the dimension. Intuitively, this can be explained by the fact that the diversity within the distribution of Gaussian process functions increases with the dimension. We believe that this falls under the broad umbrella of "Course of dimensionality" [11]. This phenomenon makes training in high dimensions very time consuming.

## 4.1 Performance on GP Test Data

We ran experiments on functions sampled from a Gaussian process with rbf kernel in different dimensions. It is important to note that these were not the functions that our model was trained on. Figure 3 shows the minimum function value found after k steps $\min_{i \leq k} f(x_i)$ for different Black-Box Optimizers. The results are averaged over 2000 different functions. Our method performs comparable to the surrogate based methods SKOPT-GP, SKOPT-GBRT and SKOPT-FOREST in low dimensions, and clearly outperforms them in higher dimensions w.r.t. minimum function value found. It looks like the Basinhopping algorithm, which is based on simulated annealing, converges much faster than all other optimizers. This however is somewhat misleading, because this algorithm is allowed to evaluate the objective functions at multiple locations for every step. We can see that the greedy LSTM optimizer based on the SUM loss performs well at the first couple of steps but then plateaus towards the end, while the more exploratory MIN loss performs worse in the beginning, but ultimately always finds the best minimum.

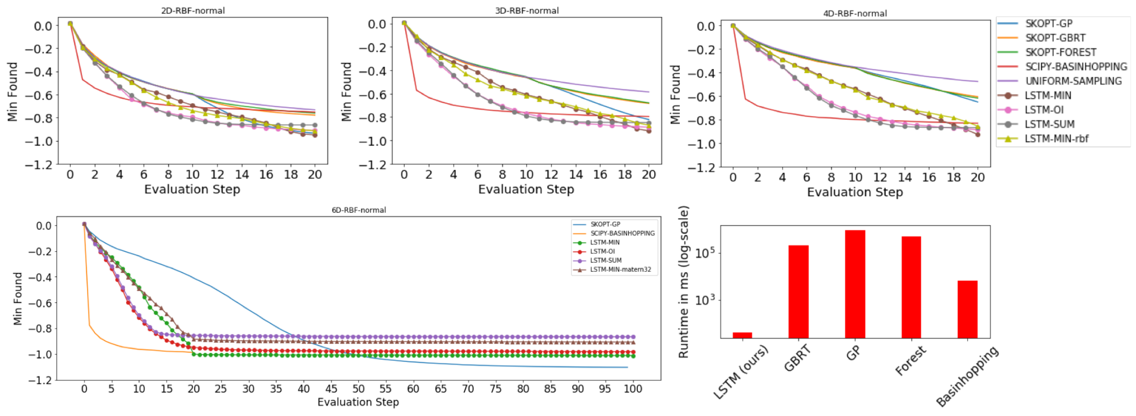

Figure 3: Top: Performance of our optimizer compared to other state-of-the-art methods on functions sampled from a Gaussian process with RBF kernel in different dimensions. Bottom left: Performance of our method in 6D over 100 steps, when trained only for 20 steps. Bottom right: Wall time taken in milliseconds by our approach and other optimization algorithms to propose 20 sampling points for 100 different 2D objective functions (Computed on a Intel® Core$^{\mathrm{TM}}$ M-5Y10 CPU, 4 cores @ 0.8 GHz, 8GB Memory).

Figure 3 also shows minimum function values found for sample sequences of 100 points in 6D. Again the results were averaged over 2000 functions sampled from a Gaussian process with RBF kernel. In this case however, our model was trained only for a sequence length of 20 steps. One can see that almost no improvement is made by the model after those initial 20 steps. This indicates that the model only works well for as many steps as it has been trained on. There are two problems with that. First, one has to decide at training time on how many steps one wants to later evaluate the model. Second, the training time increases linearly with the sequence length, making training for long sequences very time-consuming. Last, and most important, it becomes more and more difficult to train the model the longer the sequence gets. This is due to vanishing and exploding gradients mentioned earlier, that prevent reliable gradient back propagation. Although LSTMs were designed specifically to reduce this problem, it is still prohibitive to train for very long sequences. We were only able to train the model for sequences of about 20 steps. Chen et al. propose a way to work around this by gradually increasing the sequence length. Using this trick they are able to train for around 100 steps. Still training for arbitrarily long sequences is prohibitive.

The bar plot at the bottom right of Figure 3 shows that our method is up to several orders of magnitude faster than all of it's competitors. This can be explained by the fact that only matrix multiplications and simple nonlinear activations have to be computed during evaluation, while Bayesian optimization methods have to compute an expensive surrogate model. It should be noted in this context, that in many practical problems evaluation of the Black-Box function itself is very time-consuming. In these cases the runtime of the optimizer becomes negligible in relation.

## 4.2 Comparison of Priors

In this experiment we explore the effects of incorporating prior knowledge into our model. We train a model on simple quadratic functions sampled from

$$q_X(x) = \min(\|x - X\|_2^2 - 1, 1), \quad X \sim \mathcal{U}([-1,1]^d) \tag{10}$$

and compare it's performance on different types of objective functions to our other models that were trained on Gaussian process functions with rbf and matern32 kernel. The results are shown in Figure 4. As expected every model performs best on data sampled from the same distribution that it was trained on. The differences are more apparent for the 6 dimensional problems. These experiments indicate, that incorporating prior knowledge into the model by choosing a training function distribution that is similar to the objective function can lead to improved performance.

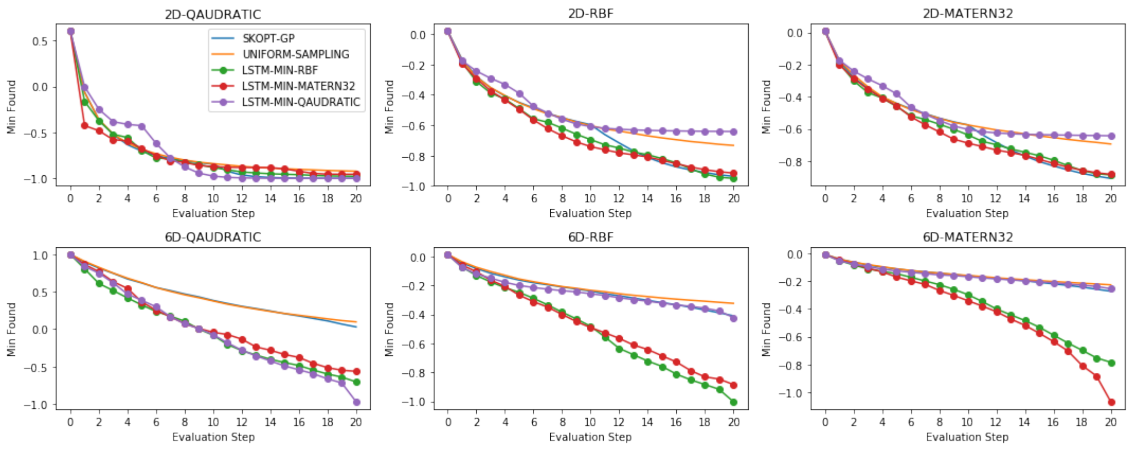

Figure 4: Comparison of different optimizers on quadratic functions (left) and functions sampled from a GP distribution with RBF kernel (middle) and Matern32 kernel (right) in dimensions 2 and 6. Average over 2000 objective functions each.

## 4.3 Airfoil Optimization

We applied our model to the real-world problem of airfoil optimization. The goal here is to find an airfoil, which maximizes the lift-over-drag quotient for stream of air from a given angle of attack. The lift and drag generated by a certain airfoil are computed using the XFOIL simulation software [4]. The shape of the airfoil is determined by a set of control points as illustrated in Figure 5. In our experiments we fixed the position of all control points, except for two which we varied along the vertical direction, thus creating a two dimensional optimization problem. A heatmap plot of one such possible 2D objective function is shown in Figure 5. It was created by performing grid search over the space of possible airfoils. The white points in the plot correspond to airfoils that are invalid. For these, the simulator just returns a very low default value. We observed that the absolute function values in this problem are always bound by 100 and used this to normalize the objective. We compared the performance of different Black-Box optimizers on the normalized airfoil optimization problem, Figure 5 bottom right. Unfortunately it seems like this benchmark is not very informative, since none of the tested optimizers perform better than random sampling. This might be explained by the many discontinuities in the objective function due to invalid airfoils. The bottom left plot shows results for the same objective functions, only this time no normalization is applied. While this has no effect on all other optimizers, our learning based method now performs significantly worse and is even outperformed by random sampling. This behavior agrees with the theoretical considerations and implies that the method should only be applied to normalizable Black-Box functions.

In another experiment we allowed all 6 control points to vary along the vertical axis, but only locally around a valid airfoil. We chose the size of this local search-space in such a way that it did not contain any invalid airfoils. The resulting 6 dimensional optimization problem is much more regular. As can be seen in Figure 6, our models benefit strongly from this increased regularity. They both perform very well on this problem. We suspect that this is because the objective function is now more similar to the training distributions of our models.

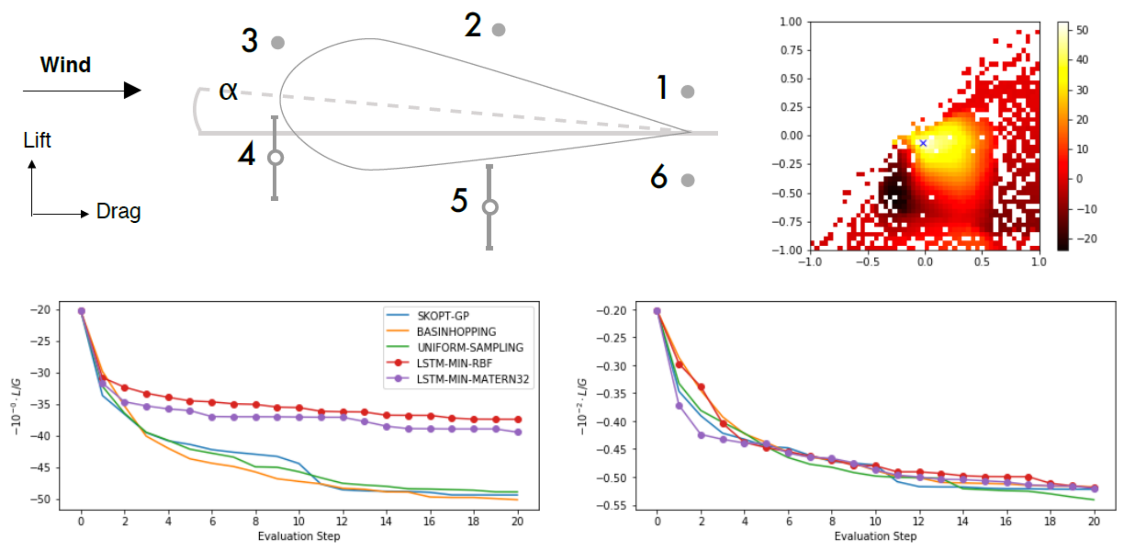

Figure 5: Top left: Sketch of the airfoil optimization problem. Top right: Heatplot of one exemplary airfoil Black-Box function found by grid search. Bottom: Performance of different Black-Box optimizers on the unnormalized (left) and normalized (right) airfoil problem. Average over 100 different combinations of control points and angles of attack.

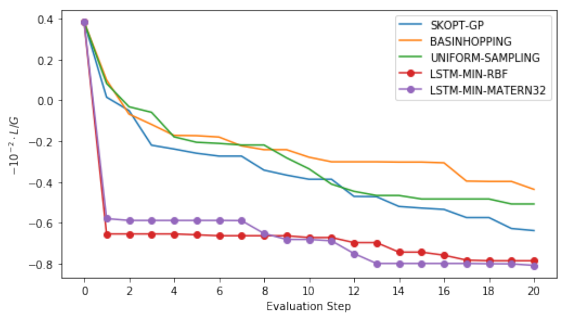

Figure 6: Performance of different Black-Box optimizers on Airfoil optimization problem in 6 dimensions. Average over 10 different angles of attack.

## 4.4 Hyperparameter Optimization

Hyperpameter optimization is an important, but time-consuming, part of designing learning algorithms. The relation between the choice of hyperparameters and the performance of the learning algorithm can be interpreted as a Black-Box function. In this experiment, we trained a Support Vector Machine with rbf kernel [3] to classify hand-written digits from the MNIST dataset [14]. We used our model to search for values of hyperparameters that yield the highest classification accuracy. More concretely we optimized for the kernel width $\gamma$ and the outlier penalty $C$. The inputs were both transformed exponentially to map to a bigger range of values. No output normalization was necessary, since the accuracy is already bound by $[0, 1]$. Due to limited computational resources, we trained the SVM only on a small subset of the original data, containing 2800 data points. We conducted a 3-fold cross-validation to estimate the accuracy obtained after training with any combination of $\gamma$ and $C$. Figure 7 shows the average negative accuracy found by different optimizers, along with the ground truth for one subset of training data found by grid search. At the end, all optimizers reach the same level of accuracy (except for Basinhopping). This might be explained by the relatively sharp transition in accuracy values for that particular range of inputs. Experiments in higher dimensions would be necessary to come to informed conclusions on this problem, but were unfortunately impossible for us to perform due to a lack of computational resources.

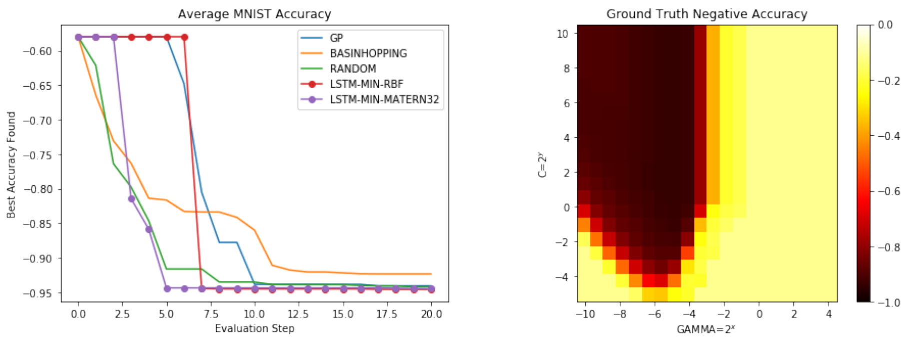

Figure 7: Left: Negative accuracy found for different optimizers over evaluation step. Average across 5 runs with different subsets of training data. Right: Ground truth of negative accuracy found by grid search. Inputs were transformed exponentially via $\gamma = 2^{x_\gamma}; x_\gamma \in [-10, 4]$ and $C = 2^{x_C}; x_C \in [-5, 10]$.

## 5 Conclusions

Our experiments support the claims made by Chen et al. and verify that their conceptually simple, learning based optimizer can compete with state-of-the art optimization algorithms on a variety of synthetic benchmarks, while outperforming them with respect to computation speed by several orders of magnitude. We showed that the model learns by itself to trade-off exploration and exploitation during sampling, and that we can guide this behavior by the choice of loss function used during training. We found evidence for the fact that incorporating prior knowledge into the model by augmenting the training distribution improves the performance of the optimizer. There are however a couple of issues with the model, that might limit the range of potential applications. First, the amount of training data needed to achieve good generalization increases super-linearly with the function dimension. Second, the model can only be trained well on normalized functions. As a result, it can only be applied to Black-Box functions for which lower and upper bounds to the function values are known. Third, direct training is only possible for sequences of about 20 steps. While this number can be raised to about 100 steps when using a curriculum to increase the sequence length as described in [2], training for sequences of arbitrary length is still impossible. In addition, the training time per sample also increases linearly with the sequence length due to the linear growth in the number operations needed to perform forward and backward pass. These issues become particularly prominent when considering the fact that the model has to be retrained for every new dimension, loss-function or number of search-steps. Based on these considerations we recommend to use the method on problems of medium dimension, for which upper and lower bounds are known, and in situations where computation time is crucial.

## 6 Acknowledgements

We want to thank Yutian Chen for his helpful and exhaustive remarks on the topic, Ashish Kumar for helping us set up the airfoil optimization problem, and NVIDIA for providing us with the computational resources to run our experiments.

## References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

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

[3] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[4] Mark Drela. Xfoil: An analysis and design system for low reynolds number airfoils. In *Low Reynolds number aerodynamics*, pages 1–12. Springer, 1989.

[5] Alexander IJ Forrester and Andy J Keane. Recent advances in surrogate-based optimization. *Progress in Aerospace Sciences*, 45(1):50–79, 2009.

[6] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[7] Joel Grus. *Data science from scratch: First principles with Python.* " O'Reilly Media, Inc.", 2015.

[8] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[11] Eamonn Keogh and Abdullah Mueen. Curse of dimensionality. In *Encyclopedia of Machine Learning*, pages 257–258. Springer, 2011.

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

[13] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[14] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[15] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[16] Carl Edward Rasmussen and Christopher KI Williams. Gaussian processes in machine learning. *Lecture notes in computer science*, 3176:63–71, 2004.

[17] Luis Miguel Rios and Nikolaos V Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.

[18] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[19] Jasper Snoek, Kevin Swersky, Hugo Larochelle, Michael Gelbart, Ryan P Adams, and Richard S Zemel. Spearmint: Bayesian optimization software, 2014.

[20] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015.