Amortized Bayesian Optimization over Discrete Spaces

Yulia Rubanova[*]
University of Toronto
Vector Institute

David Dohan
Google Research

Kevin Swersky
Google Research

Kevin P. Murphy
Google Research

Abstract

Bayesian optimization is a principled approach for globally optimizing expensive, black-box functions by using a surrogate model of the objective. However, each step of Bayesian optimization involves solving an inner optimization problem, in which we maximize an acquisition function derived from the surrogate model to decide where to query next. This inner problem can be challenging to solve, particularly in discrete spaces, such as protein sequences or molecular graphs, where gradient-based optimization cannot be used. Our key insight is that we can train a parameterized policy to generate candidates that maximize the acquisition function. This is faster than standard parameter-free search methods, since we can amortize the cost of learning the policy across rounds of Bayesian optimization. We therefore call this Amortized Bayesian Optimization. On several challenging discrete design problems, we show this method generally outperforms other methods at optimizing the inner acquisition function, resulting in more efficient optimization of the outer black-box objective.

1 INTRODUCTION

Many applications involve finding a structure or input that maximizes a black-box target function that is expensive to evaluate. For example, in protein sequence design, the goal is to construct a protein that binds to a specific target; evaluating the binding affinity is noisy and time consuming, since in this case it involves synthesizing the protein in a lab.

[*] Work completed during an internship at Google.

Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence (UAI), PMLR volume 124, 2020.

Bayesian optimization (BO) is a common approach to optimizing a black-box target function when only a limited number of evaluations can be used (Mockus et al., 1978; Jones et al., 1998; Shahriari et al., 2015). BO constructs a surrogate model $\mathcal{M}$ that approximates the true function (also known as the "oracle") and provides uncertainty estimates. Using $\mathcal{M}$, BO computes an *acquisition function* (AF) that balances exploration (searching new areas) and exploitation (making local improvements) of the search space. In each iteration, BO optimizes the current acquisition function (the "inner loop" optimization problem), returning a set of recommendations to be evaluated in the next round. These are evaluated by the oracle, and the (string, reward) pairs are added to a dataset $\mathcal{D}$. $\mathcal{M}$ is then re-trained on $\mathcal{D}$, and the cycle repeats until a budget on the number of oracle evaluations is reached.

While BO has found numerous applications in continuous domains (e.g., Snoek et al. (2012)), it is less prominent in discrete problems, such as protein sequence design. There are arguably two main reasons for this: the lack (until recently) of good Bayesian regression models for string inputs, and the difficulty of solving the inner AF. The former problem has been addressed, by using techniques from Bayesian deep neural networks (we use an ensemble of DNNs), so we focus on the second issue: efficiently optimizing the inner AF.

Evolutionary solvers (ES) are commonly used for black-box optimization over string functions when the function itself can be cheaply evaluated (Real et al., 2019; Wu et al., 2019), as is the case with an acquisition function. The main idea is to iteratively refine a population of strings by randomly perturbing them and then selecting the individuals with higher reward. The classic evolutionary algorithms choose edits randomly at each iteration and rely on a selection procedure to find the best strings. However, this stochastic hill-climbing approach can be prone to finding sub-optimal solutions and often requires many iterations.

Algorithm 1 Bayesian Optimization

```
Input: Black-box reward function $r(s)$
$\mathcal{P}^0 \sim p(s)$ {Sample initial population}
$r^0 \leftarrow \{r(\mathcal{P}^0)\}$ {Evaluate rewards}
$\mathcal{D} \leftarrow \{\mathcal{P}^0;\ r^0\}$ {Make a dataset}
for $j = 1$ to $T_{outer}$ do {Outer loop}
  Train predictor model $\mathcal{M}$ on dataset $\mathcal{D}$
  Define acquisition function $AF_j$ using predictor $\mathcal{M}$
  $\mathcal{P}_j \in \text{argmax}_{s \sim p(s)} AF_j(s)$ {Solve the inner loop}
  $r^j \leftarrow \{r(\mathcal{P}^j)\}$ {Evaluate rewards}
  $\mathcal{D} \leftarrow \mathcal{D} \bigcup \{\mathcal{P}^j;\ r^j\}$ {Add new strings to the dataset}
end for
Return Candidates from the last batch $\mathcal{P}_{T_{outer}}$
```

In this work, we exploit the simple observation that the sequence of optimization problems in the inner loop of BO are closely related to each other. While typically each inner loop is solved from scratch, our approach instead learns efficient strategies for optimizing the acquisition function by learning from previous acquisition functions. We propose the Deep Evolution Solver (DES), where we use a neural network to suggest edits to apply to a given string, and train this network by policy gradient-based reinforcement learning. We train the policy online: simultaneously evolving the population of strings and optimizing the parameters of the policy. This allows the network to be both efficient and adaptive: quickly honing in on good solutions while also adjusting based on new data that characterizes the current acquisition function.

We use DES in the inner loop of BO and apply it to real-world tasks of protein energy modelling. While our primary aim is to improve the efficiency of the inner loop, we find that DES often proposes better queries for the outer loop as well, leading to solutions with higher reward. Overall, DES is a promising step towards improving BO for discrete black-box functions. We release our implementation at `https://www.github.com/google-research/google-research/tree/master/amortized_bo`.

2 BACKGROUND

2.1 PROBLEM SETUP

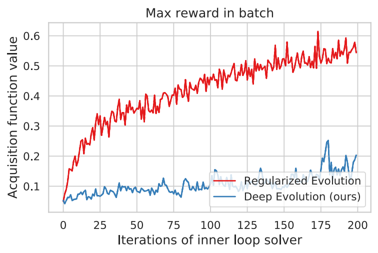
Our goal is to find $s^* = \arg\max_{s \in \mathcal{S}} f(s)$, where s is a string of length L on an alphabet of size A, and $f : \mathcal{S} \rightarrow \mathbb{R}$ is an unknown black-box objective. We are interested in the batch optimization setting, so at each round, we generate a population of strings $\mathcal{P} = \{s_i\}_{i=1..P}$, where P is the population size. Our goal is to create a population of strings (either by editing an initial population, or generating from scratch) such that one or more members have high fitness, using as few calls to the f function as possible.

2.2 BAYESIAN OPTIMIZATION

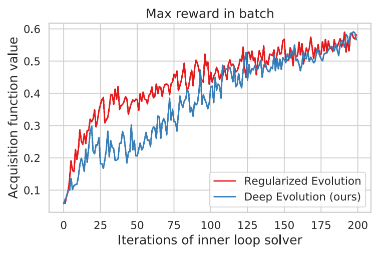
The aim of BO is to globally maximize a noisy black-box reward function in as few evaluations of the function as possible. It involves building a surrogate model $\mathcal{M}$ that approximates the true function and is relatively cheap to evaluate. It proceeds by sampling the initial population of strings and getting their rewards, next it constructs a predictor to approximate the true reward function. BO uses a Bayesian predictive model to define a posterior distribution over the rewards in order to encourage better exploration of the space. BO suggests strings by maximizing an *acquisition function* that takes in a point estimate and its uncertainty; it balances the tasks of maximizing the target value and exploring the new regions of the space. These strings are the proposed candidates to be evaluated in the next round. After evaluating new set of strings through target function, we re-compute posterior distribution over the rewards and obtain a new acquisition function. Solving the acquisition function is typically referred to as the "inner loop", and evaluating the true reward as the "outer loop".

Gaussian processes are a common choice for the predictive model in continuous domains, however they scale poorly with the amount of data and do not operate as well in high-dimensional spaces. However, following (Lakshminarayanan et al., 2017), we have found that an ensemble of neural networks works well as a substitute. To obtain a posterior distribution, we train the predictor on the dataset of strings and their rewards and compute the mean $\mu(s)$ and the standard deviation $\sigma(s)$ of the predictions.

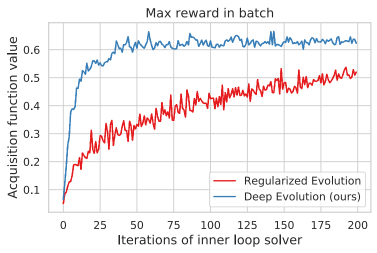
There are multiple ways to define the acquisition function. In this paper, we consider Upper Confidence Bound (UCB) (Srinivas et al., 2009), Thompson Sampling (Thompson, 1933), and Posterior Mean. UCB is defined as $\text{AF}_{\text{UCB}}(s) = \mu(s) + \sigma(s)$, where $\mu(s)$ and $\sigma(s)$ are the mean and the standard deviation of the predictions. Thus, UCB favors samples with either a high predicted reward, or a large amount of uncertainty. Thompson sampling involves randomly sampling a function from the posterior and maximizing it (c.f., (Lu & Van Roy, 2017)). When we use the posterior Mean acquisition function, we simply maximize the mean $\mu(s)$ of the predictive model, without explicitly trying to explore uncertain parts of the input space.



(a) Outer loop iteration 1



(b) Outer loop iteration 2



(c) Outer loop iteration 3

Figure 1: Training curve of the inner solver in the first three outer loop iterations of BO on the length-50 Protein 5P21 problem. While not effective initially, after a few iterations ABO becomes highly efficient at solving the inner loop.

2.3 EVOLUTIONARY ALGORITHMS

Here we outline several evolutionary algorithms that we use as baselines both within and outside of Bayesian optimization. Evolutionary algorithms proceed by repeatedly perturbing and subsampling from a population of candidate solutions. While certainly not exhaustive, the baseline methods we list below have been shown to be highly effective on the problems we consider (Belanger et al., 2019).

Single Mutant Walker Single Mutant Walker (Wu et al., 2019) is a hill-climbing algorithm that takes the best string from the previous batch and constructs all possible single-character mutants of that string. If the resulting population is larger than the population size, it randomly subsamples the mutants to get back to the required population size. This new population is then evaluated on the reward function.

Regularized evolution To create a new string, Regularized evolution (Real et al., 2019) selects two parent strings from the population, performs cross-over between them, and then mutates the string. This process is repeated to create each string for the next population. Parents are selected via tournament selection by sampling a subset of 10 strings from the population without replacement and selecting the string with highest reward. To perform the cross-over, we copy one of the parent strings, switching between two parents with probability 0.1. Finally, each position in the string is mutated with probability 0.1. The oldest strings in the population are removed from consideration.

Note that Single Mutant Walker and Regularized evolution use selection to improve the average reward. Regularized Evolution also uses cross-over, which allows it to swap a potentially large part of the string with a single move. Our DES method uses a much simpler, policy-based local search algorithm to maximize the acquisition function, with only simple character-level mutations; expanding the set of possible mutations used by our solver is an interesting topic for future work.

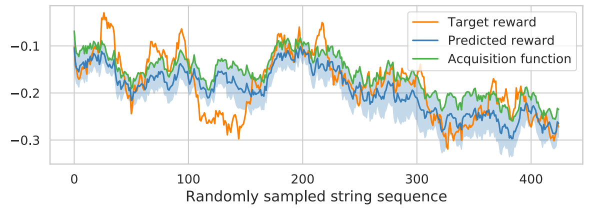


Figure 2: Example of a target function, reward predicted by the ensemble of neural nets, and Upper Confidence Bound acquisition function for Protein Ising model problem. The x-axis show the randomly sampled sequence of strings that differ by one character. All the functions are highly multi-modal, i.e., they have sharp changes indicating that strings that differ by a few characters can have drastically different rewards.

3 METHODS

Here we discuss the idea of Amortized Bayesian Optimization, where we re-use the parameters of the inner solver for the subsequent iterations of BO. Since our learned solver is based on an evolutionary algorithm, which predicts the edits (mutations) for each string in a population, we call it the deep evolutionary solver (DES). The inner loop is shown visually in Figure 3. For the j^{th} outer loop step, once the acquisition function is maximized, the resulting population of strings $\mathcal{P}^j$ is evaluated on the true objective and added to the dataset $\mathcal{D}$, which is used to train the Bayesian surrogate model (an MLP ensemble in this case).

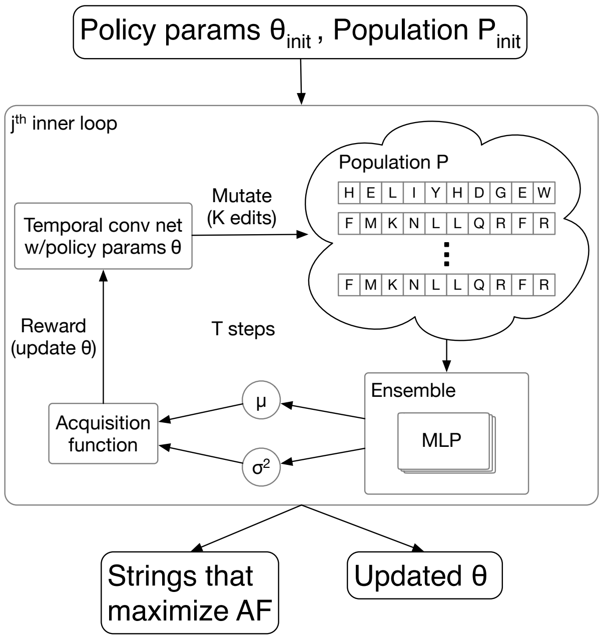


Figure 3: An illustration of the DES inner loop. DES iteratively mutates a population according to a policy network. The network parameters θ are updated online via REINFORCE. The strings that maximize the acquisition function are used to update the surrogate, and the updated parameters are used to initialize the policy network for the next round.

3.1 DEEP EVOLUTION SOLVER

Taking inspiration from evolutionary algorithms, we construct a solver that continually evolves a population of strings. On a high level, the solver works as follows. We start with a population of randomly sampled strings $\mathcal{P}^0 = \{s_i \sim \mathcal{A}^N\}_{i=1..P}$. Then, we perturb the strings with edits sampled from a policy network and obtain the next string population $\mathcal{P}^1 = \{s'_i\}_{i=1..P}$. Finally, we run the strings through an oracle to obtain the new rewards $\{r(s')\}_{s' \in \mathcal{P}^1}$. We repeat the process for T iterations.

Policy network We propose to use a policy network $\pi_\theta(s)$ to predict a distribution over edits to apply to a given string. The network perturbs a string by sequentially sampling K edits from the policy. Thus, an action is a sequence $a = \{e_k\}_{1:K}$, where each e_k specifies a location and a desired letter. The likelihood of an action is given by the distribution $\pi_\theta(a|s) = \prod_{k=1}^{K} \pi_\theta(e_k|s, e_1...e_{k-1})$. Since we sample K edits sequentially, we allow the network to edit the same position twice and to set the position to the same character as in the original string. Therefore, the string will make at most K edits to the string, but could make fewer. Although we use single-character edits in this work, the framework is agnostic to the specific mutation functions over a string (e.g., transpositions).

Algorithm 2 Deep Evolution Solver

```
**Input:** Acquisition function $F(s)$
Randomly sample string population $\mathcal{P}_1 = \{s_i\}$
Evaluate fitness of the strings $\{F(s_i)\}$
**for** $t = 1$ **to** $T_{\text{inner}}$ **do**
  **for all** $s_i \in \mathcal{P}_t$ **do**
    $s'_i = s_i$
    **for** $k = 1$ **to** $K$ **do**
      $e^i_k \sim \pi_\theta(e^i_k|s'_i)$ {Sample edit from the policy}
      $s'_i = \text{Mutate}(s'_i, e^i_k)$ {Apply edit to the string}
    **end for**
    Let $a_i = (e^i_1, ...e^i_K)$
    Let $p_i = \pi_\theta(a_i|s_i) = \prod_{k=1}^{K} \pi_\theta(e^i_k|s_i)$
    $R(s_i, a_i) = F(s'_i) - F(s_i)$ {Fitness improvement}
  **end for**
  $P_{t+1} = \{s'_i\}$ {Form a new string population}
  $\nabla_\theta \mathcal{L} = \frac{1}{|\mathcal{P}_t|} \sum_{i=1}^{|\mathcal{P}_t|} [R(s_i, a_i) \nabla_\theta \pi_\theta(a_i|s_i)]$
  $\theta = \theta - \alpha \nabla_\theta \mathcal{L}$ {SGD on policy parameters}
**end for**
Sort and return top $P$ strings from $\mathcal{P}_{1:T_{\text{inner}}}$
```

After applying the edits to a a batch of strings, we evaluate the rewards of the new strings and update the parameters of the the policy using policy gradient, using the difference in score between the perturbed child and the parent as the reward . Then, we continue to mutate the same population of strings in an online fashion. Evolving the population and training the policy simultaneously has a number of advantages over episodic training. First, online training requires a single run of the evolutionary solver and has comparable run-time and number of calls to the reward as the stochastic solvers. In contrast, previous approaches such as (Schuchardt et al., 2019) use episodic training, requiring that the solver is re-run many times to train the policy. Second, if using the solver in a stand-alone fashion, we don't need to determine the number of steps in advance.

REINFORCE gradient estimator We aim to improve the average reward of the strings in the batch by maximizing the following function:

$$\mathcal{L}(\theta) = \mathbb{E}_{s \sim p(s)} \mathbb{E}_{a \sim \pi_\theta(a|s)}[R(s, a)] = \frac{1}{|\mathcal{P}|} \sum_{i=1}^{|\mathcal{P}|} \mathbb{E}_{a \sim \pi_\theta(a|s_i)}[R(s_i, a)] \quad (1)$$

where $R(s_i, a) = F(s'_i) - F(s_i)$; $s'_i = \text{mutate}(s_i, a)$ is the string perturbed using action a; $F(s)$ is the current acquisition function; and $p(s)$ is the distribution over strings.

We use the REINFORCE (Williams, 1992) gradient estimator to update the parameters of the policy:

$$\nabla_\theta \mathcal{L}(\theta) = \frac{1}{|\mathcal{P}|} \sum_{i=1}^{|\mathcal{P}|} \mathbb{E}_{a \sim \pi_\theta(a|s_i)} [R(s_i, a) \nabla_\theta \pi_\theta(a|s_i)] \tag{2}$$

Here, we used the fitness of the original string $F(s_i)$ as a baseline for the REINFORCE estimator: $R(s_i, a) = F(s'_i) - F(s_i)$. By computing the difference between the original and perturbed strings, we force the model to learn the edits to improve the fitness of each individual string. We found the method to work consistently across different problems without the addition of a learned baseline as in actor-critic methods.

Modelling string edits We define an edit as a combination of the position in the string and the new character to place into this position. We model the distribution over edits as the following joint distribution:

$$\pi_\theta(e|s) = \pi_\theta(\text{char}, \text{pos}|s) = \pi^c_\theta(\text{char}|\text{pos}, s)\pi^p_\theta(\text{pos}|s) \tag{3}$$

Both distributions $\pi^c_\theta(\text{char}|\text{pos}, s)$ and $\pi^p_\theta(\text{pos}|s)$ are modelled as categorical distributions. We factorize the distribution instead of modelling the joint distribution directly to avoid taking the softmax over a potentially big search space $L \times |\mathcal{A}|$, where L is the length of the string and $|\mathcal{A}|$ is the size of the alphabet.

Policy network implementation We implement the policy as a convolutional neural network that takes a population of the strings in a one-hot encoding with dimensionality $|\mathcal{P}| \times L \times |\mathcal{A}|$. The network output consists of two parts: logits for distribution over positions of size $|\mathcal{P}| \times L$ and logits for distribution over characters generated separately for each position of size $|\mathcal{P}| \times L \times |\mathcal{A}|$.

Selection Typically, evolutionary algorithms perform an additional step of selecting the set of best strings from the previous population. In contrast, we do not use selection and rely on a learned policy to suggest edits that will improve the reward for each string.

3.2 AMORTIZED BAYESIAN OPTIMIZATION

Surrogate model For our Bayesian surrogate model, we use an ensemble of 10 shallow feed-forward neural networks, similar to Belanger et al. (2019). We use mean of the ensemble as an estimate for the reward and the standard deviation for the uncertainty estimate.

Algorithm We begin with an initial population of strings and evaluate them with the true reward function. This population is randomly sampled from the uniform distribution over the character alphabet. For each problem, we use the same starting population for all BO runs to make a fair comparison of the inner solvers. We train the neural net ensemble on the starting population and their rewards and obtain the acquisition function for the first iteration.

Next, we run the inner solver to find the string that maximizes the acquisition function. The inner solver is initialized with a random population each time. At the end of the inner loop, we sort the strings from the entire run of the evolutionary solver by the value of the acquisition function and use the best strings as the candidates for the next iteration of BO (without duplicates). We evaluate the candidates with the true reward, and add them to the dataset. In the next iteration of the outer loop, we re-train the regressor on the updated dataset and solve a new acquisition function.

Amortization We implement Amortized Bayesian Optimization, where we re-use the policy trained in the previous iterations of the outer loop for the next iteration. We apply this trick to the policy weights of DES and refer to it as “warm-start”. We compare it to DES where the policy network parameters are randomly initialized after every outer loop iteration and refer to this as “cold-start”.

4 EXPERIMENTAL RESULTS

4.1 DATASETS

We test our approach on two kinds of string optimization problems, motivated by protein sequence design. In both cases, the length L is fixed, and ranges from 20 to 100, and the alphabet size A is fixed at 20.

Alternating Chain We initially consider a synthetic problem of generating a string of alternating numbers. The string with the highest reward consists of exactly two numbers alternating with each other. Despite its simplicity, this problem is difficult for evolution-type solvers, since it is easy to find substrings of alternating numbers, but it is hard to find a global solution. We experiment with string lengths 20, 50, and 100, and alphabet size 20.

Protein Contact Map Potts Model We also apply our approach to finding a protein sequence that minimizes the folding energy. The energy is computed by a Potts model derived from contact maps in the Protein Data Bank (Berman et al., 2000), as in (Belanger et al., 2019). We test on string lengths 20, 50 and 75 on Protein 5P21. See 4 for an illustration of the structure of the Potts model.

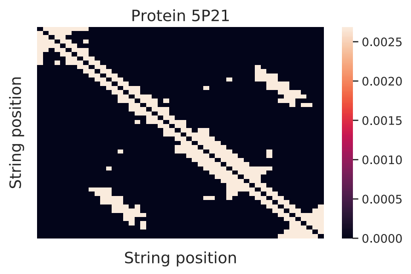


Figure 4: Example of a contact map used in Protein Ising model problem: 5P21 (length 50).

4.2 EXPERIMENT SETUP

We evaluate the performance of the baseline evolutionary algorithms (EA) when used as stand-alone solvers, and compare them to BO; in the BO case, we consider using these baseline EA solvers for the inner loop, as well as cold-start and warm-start versions of our DES solver. (We do not use DES as an outer loop solver, since it is too sample inefficient.) We assume that calling the oracle is expensive, and set a fixed budget on the number of oracle evaluations for all solvers. Thus, we compare stand-alone solvers only for the same number of iterations as we used for the BO outer loop.

We focus on the scenario when we don't have any information about the problem in advance, meaning that we cannot use pre-trained models. Instead, in DES we train the policy online, together with evolving the population. To ensure fairness with the baselines, each solver begins with the same initial population of strings.

4.3 TRAINING DETAILS

We used the same hyperparameters for all problems, varying only the number of edits per step of DES. We used $K = 10$ edits per step for length $L = 20$ strings, $K = 20$ for $L = 50$, and $K = 70$ for $L = 75$ and $L = 100$. We re-run each experiment with three random seeds and show the mean and standard deviation across multiple runs.

We use a population size of 500 in both the inner loop and the outer loop. As a first step in the BO experiments, we randomly sample a set of 500 strings and train an ensemble of regressors on this set. Then, we run the inner solver, which maintains a population of 500 strings that are mutated over 300 steps of the inner loop. As a last step of the inner solver, we choose the best 500 samples under the surrogate throughout the entire course of the inner solver run. We evaluate these candidates with the oracle function, add these to the set of evaluated strings, and repeat the BO process. In total, we perform 15 iterations of the BO outer loop. To match the number of oracle calls across methods, we run stand-alone solvers for 15 iterations. (Thus the total number of oracle calls is $15 \times 500 = 7500$.)

The uncertainty estimates come from an ensemble of ten feed-forward networks of three layers (32, 8, 4 units) with ReLu activations. In every iteration of the outer loop, we train an ensemble for 10 epochs with batch size of 50 and learning rate 0.01 with mean squared error loss.

For DES, we use a policy network of one convolutional layer. We used a depth 300, a 1D kernel of size 5, and a stride of 1 applied along the positions in the string. The convolution is followed by a ReLU and a linear layer to compute the policy. We add positional encoding (Vaswani et al., 2017) to the one-hot representation of the input string. We found that it is important to preserve the positional information of the string, since we use the policy to predict the distribution over characters for each position separately. We use the ADAM (Kingma & Ba, 2015) optimizer with learning rate 10^{-3}.

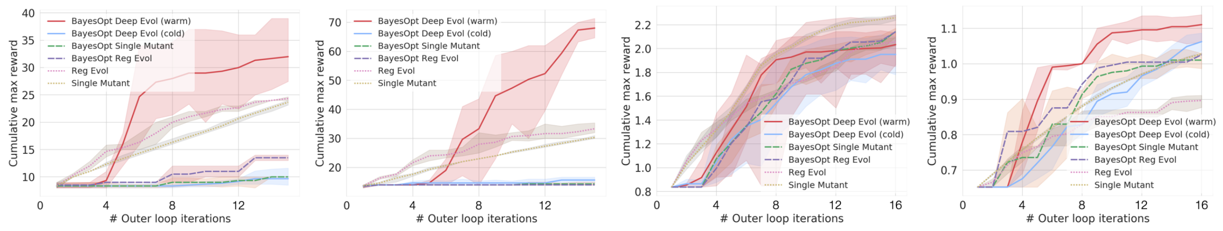
4.4 COMPARISON WITH OTHER SOLVERS

Fig. 5 demonstrates results on the Alternating Chain and Protein Contact Map problems using the Thompson Sampling acquisition function. We see that BO is more sample efficient than applying EA directly to the objective, and that our warm-started Deep Evolution Solver inside of BO often outperforms, or is competitive with other solvers. More experiments across a range of settings can be found in the supplementary material. Tables 1 and 2 demonstrate the results on other string lengths and for other acquisition functions. Generally, DES performs better than other methods for string lengths 50 and higher. On smaller problems (length 20), our approach is comparable to other solvers. We additionally include results of the deep evolution solver run against a Gaussian process regressor as an alternative to an ensemble. At each round, we select GP hyperparameters using 5 fold cross validation across RBF, Matérn, and quadratic kernels. We find that GPs do not scale well to our setting given the large number of data points (10,000 over the course of training), and focus on the faster ensemble approach.

4.5 ABLATIONS

Fig. 6 shows the performance for a range of inner-loop steps on the Protein length-50 problem. As expected, the cold-start policy gradient and regularized evolution inner-solvers improve with more steps. With DES, while it reaches the highest overall performance, it degrades when allowed too many (1000) steps. This suggests that without further regularization, it can overfit to earlier iterations.

Fig. 7 varies the number of edits K that we sample from the policy for each string in one iteration of the inner loop



(a) Alt Chain 50 (b) Alt Chain 100 (c) Protein 20 (d) Protein 50

Figure 5: Cumulative maximum reward throughout the training on (a-b) Alternating Chain problem, and (c-d) Protein problem. Red is BO with our warm-started DES solver.

Alternating chain	Length 20			Length 50			Length 100		
Deep Evol: # edits	10			30			70		
Acquisition function	UCB	Thmpsn	Post	UCB	Thmpsn	Post	UCB	Thmpsn	Post
Single Mutant	14.67	**14.67**	14.67	23.67	23.67	23.67	30.33	30.33	30.33
Regularized Evol	14.67	**14.67**	14.67	24.33	24.33	24.33	33.33	33.33	33.33
BO + Single Mutant	15.33	8.00	13.67	19.33	8.66	22.00	15.00	15.00	15.67
BO + Regularized Evol	16.67	10.50	15.00	30.67	14.00	35.00	22.67	14.67	25.00
BO + Deep Evol (cold)	**18.00**	12.67	**19.00**	33.00	17.33	26.33	18.67	22.50	13.67
BO + DES	16.67	14.00	16.33	**42.00**	**41.33**	**39.00**	**78.33**	**47.67**	41.00
BO GP + DES	13.67	-	16.34	32.67	-	24.67	33.67	-	**42.67**

Table 1: Max cumulative reward for Alternating Chain averaged over three runs with different random seeds. Bayesian optimization solvers are performed with 300 steps in the inner loop and 15 iterations of the outer loop. Single Mutant Walker and Regularized Evolutation baselines were run for 15 iterations to match the BO experiments.

on (a) the Alternating Chain length-100 problem (b) the Protein length-50 problem. More edits per step allows the policy to make larger moves, which helps on the Protein problem, however it can also make the optimization problem more difficult as demonstrated on the Alternating Chain problem, since it has to take noisy policy gradients through a sequence of many edits.

More results on these experiments can be found in the supplementary material. Ultimately, this shows that a fruitful area of future exploration would be to find robust regularizers and mutation types for the inner-loop solvers.

4.6 BENEFITS OF AMORTIZATION

In this section, we demonstrate why Amortized BO with warm-started solver is beneficial. Fig. 1 compares inner-loop optimization using DES and Regularized Evolution for the first three rounds of BO on the length-50 Protein 5P21 problem. In the first outer loop iteration, DES starts with a randomly initialized policy and therefore improves the reward more slowly than Regularized Evolution. In the next iteration, warm-started DES and Regularized Evolution optimize the acquisition function at roughly the same rate. In the third iteration, DES reaches a high-valued solution of the acquisition function in less than 50 iterations, while Regularized Evolution fails to achieve a similar value after 200 iterations of the inner loop. This shows that the warm-started policy can be an efficient optimizer for subsequent iterations, reducing the number of inner loop steps required to solve the acquisition function.

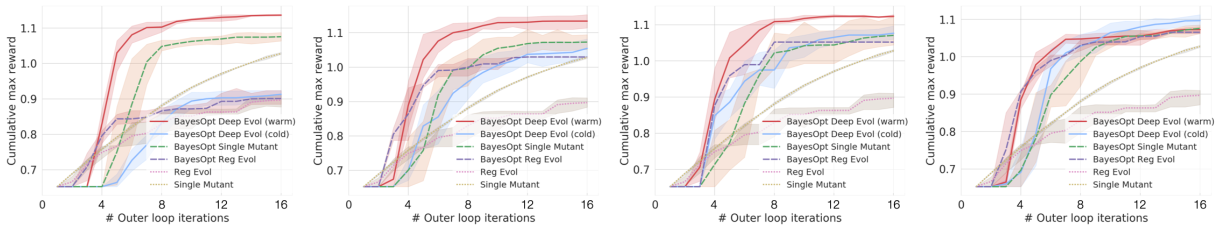
4.7 BENEFITS OF A LEARNED INNER-LOOP SOLVER

Successful inner-loop optimization can correlate with outer-loop efficiency. As shown in Section 4.4, DES can also improve the outer loop, finding samples with higher reward. However, in many cases the cost of evaluating the acquisition function is assumed to be negligible in comparison to evaluating the true reward. A natural question therefore is whether one can simply run an evolutionary solver for a large number of iterations and reach the same reward. In other words, does DES improve optimizer efficiency alone, or can it also find better solutions to the acquisition function? To test this we run Single Mutant Walker for up to 10,000 iterations in the inner loop.

As shown in Fig. 8a, on the Protein problem, Single Mutant Walker achieves the approximately same reward regardless of the number of steps in the inner loop, while DES finds candidates with a higher reward. In other words, a greedy hill-climbing policy has the potential to get stuck in a local optimum. At the other extreme, pure

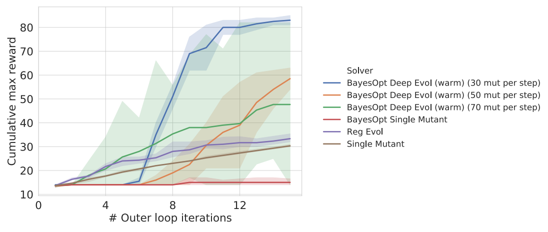
Protein Contact Map	Length 20			Length 50			Length 75		
Deep Evol: # edits	10			20			70		
Acq. function	UCB	Thmpsn	Post	UCB	Thmpsn	Post	UCB	Thmpsn	Post
Single Mutant	2.260	**2.259**	**2.259**	1.028	1.028	1.028	0.859	0.859	0.859
Regularized Evol	2.056	2.056	2.056	0.897	0.897	0.897	0.735	0.735	0.735
BO + Single Mutant	2.206	2.219	2.204	1.073	0.996	1.074	0.984	**0.994**	0.999
BO + Reg. Evol	2.238	2.154	2.205	1.030	0.989	1.032	0.812	0.809	0.817
BO + Deep Evol (cold)	2.234	2.100	2.224	1.065	1.031	1.063	0.560	0.560	0.560
BO + DES	2.186	2.175	2.254	**1.134**	**1.109**	**1.092**	**1.225**	0.812	**1.099**
BO GP + DES	**2.267**	-	2.228	0.653	-	0.891	0.560	-	0.560

Table 2: Max cumulative reward for Protein Contact map problem, averaged over three runs with different random seeds. Bayesian optimization solvers are performed with 300 steps in the inner loop and 15 iterations of the outer loop. Single Mutant Walker and Regularized Evolution baselines were run for 15 iterations to match the BO experiments.

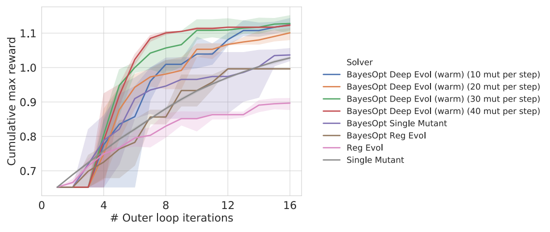


(a) 100 iterations (b) 300 iterations (c) 500 iterations (d) 1000 iterations

Figure 6: Cumulative maximum reward throughout the training depending on the number of inner loop steps on Protein Contact Map problem of length 50. For BO, we show cumulative maximum reward of the outer loop. We used Upper Confidence Bound acquisition function for these experiments.



(a) Alt Chain 100



(b) Protein 50

Figure 7: Max achieved reward in the Bayesian optimization inner loop depending on number of edits per step of Deep Evolution Solver. The line and the shaded area show mean and standard deviation over three random seeds. **(a)** Alternating chain, length 100, Thompson sampling, 300 steps of inner loop. **(b)** Protein 5P21, length 50, Thompson sampling, 300 inner loop steps.

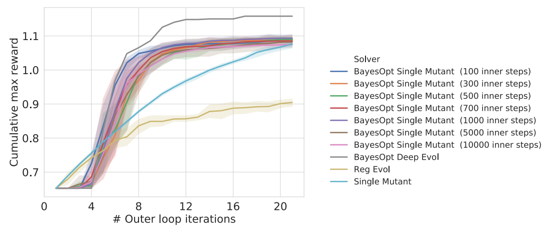
random sampling will eventually find the global optimum, but it may take an infeasible amount of time. DES has the potential to get the best of both worlds: finding high-reward solutions in a relatively short amount of time.

4.8 ACCURACY OF THE SURROGATE MODEL

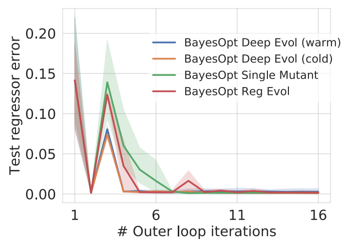
An essential determiner of BO efficiency is the ability of the regressor to fit the true reward function. Fig. 8b shows the mean squared error of the regressor from the previous iteration of BO and applied to the new candidate sequences produced by the inner loop. Starting from the 4th iteration, the regressor test error is small, indicating that the ensemble has learned to accurately generalize on the target reward function.

5 RELATED WORK

BO for discrete spaces Baptista & Poloczek (2018) identify the difficulty of applying BO to discrete combinatorial spaces. In response, they propose a new acquisition



(a) SingleMutantWalker Iterations



(b) Regressor Error

Figure 8: (a): Single Mutant Walker as the inner solver in the Bayesian optimization inner loop. Even with a large number of steps in the inner loop, Single Mutant Walker is not able to improve the reward. (b): BO regressor error on new candidate sequences propose by the inner solver.

function that is amenable to using semi-definite programming to maximize the inner-loop. Their approach is only applicable when the surrogate takes the form of Bayesian linear regression. Kandasamy et al. (2019) use a Gaussian process with hamming distance kernel to fit a surrogate on discrete spaces. They use evolutionary solvers to optimize the acquisition function. Oh et al. (2019) apply graph kernels to BO over graphs, and use a greedy hill climbing local search to maximize the acquisition function. Wilson et al. (2018) apply the reparameterization trick to maximizing acquisition functions. Kim & Choi (2019) study the regret between the acquisition function global optima and the local optima found by local optimizers, such as LGBFS, in the continuous BO setting. They note that this bound becomes tighter as the local optimizers are allowed to restart in more places.

Latent variable models for protein design Killoran et al. (2017) use a Generative Adversarial Network (GAN) to generate strings resembling existing strings with the highest fitness. Gomez-Bombarelli et al. (2018) train a variational autoencoder (VAE) to fit the available strings, and apply BO in the latent space to produce the next round of samples. In particular, they use GP regression in the latent space to predict the fitness; they then use gradient optimization in the latent space, and decode the optimal latent point back to a string. However, this can result in the VAE being asked to decode in parts of space where it is not well-trained, resulting in low-quality decodings. Brookes et al. (2019) design biological sequences using the cross entropy method, which fits a generative density model to the top sequences seen so far. They use a VAE as the generative model.

Improving evolutionary search with RL Schuchardt et al. (2019) train an RL agent to learn to mutate a string, and perform selection and cross-over. They treat the entire run of the evolutionary solver as one episode and repeat the evolutionary search 500 times before updating the policy. In contrast, our approach learns a policy from a single run of evolutionary solver, using the same number of oracle evaluations and a similar wall-clock run time as other evolutionary solvers.

Paliwal et al. (2019) predict a distribution over binary strings to produce an initial population of strings. The "mutants" are generated by sampling random strings from the same distribution and performing cross-over between newly sampled and high-performing strings in the population. In contrast, we directly learn the distribution over mutations to apply to a given string.

6 CONCLUSION AND FUTURE WORK

We have introduced Amortized Bayesian Optimization, which transfers knowledge across the sequence of inner-loop acquisition functions by learning a parameterized inner-loop optimizer online, resulting in a faster inner-loop optimization for Bayesian optimization and better solutions. The inner-loop solver (DES) uses a parameterized a transition policy for local search, but it is possible to amortize solving the inner loop using other kinds of generative models. Finding adaptable and robust inner-loop solvers is a promising area that could dramatically improve the application of Bayesian optimization to discrete spaces.

References

Baptista, R. and Poloczek, M. Bayesian optimization of combinatorial structures. *International Conference on Machine Learning*, 2018.

Belanger, D., Vora, S., Mariet, Z., Deshpande, R., Dohan, D., Angermüller, C., Murphy, K., Chapelle, O., and Colwell, L. J. Biological sequence design using batched Bayesian optimization. *NeurIPS 2019 Workshop on Machine Learning and the Physical Sciences*, 2019.

Berman, H. M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T. N., Weissig, H., Shindyalov, I. N., and Bourne, P. E. The protein data bank. *Nucleic Acids Res*, 28: 235–242, 2000.

Brookes, D., Park, H., and Listgarten, J. Conditioning by adaptive sampling for robust design. *International Conference on Machine Learning*, 2019.

Gomez-Bombarelli, R., Wei, J. N., Duvenaud, D., Hernández-Lobato, J. M., Sánchez-Lengeling, B., Sheberla, D., Aguilera-Iparraguirre, J., Hirzel, T. D., Adams, R. P., and Aspuru-Guzik, A. Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science*, 4(2):268–276, 2018. doi: 10.1021/acscentsci.7b00572. PMID: 29532027.

Jones, D. R., Schonlau, M., and Welch, W. J. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

Kandasamy, K., Vysyaraju, K. R., Neiswanger, W., Paria, B., Collins, C. R., Schneider, J., Poczos, B., and Xing, E. P. Tuning hyperparameters without grad students: Scalable and robust Bayesian optimisation with dragonfly. *arXiv preprint arXiv:1903.06694*, 2019.

Killoran, N., Lee, L. J., Delong, A., Duvenaud, D. K., and Frey, B. J. Generating and designing DNA with deep generative models. *arXiv preprint arXiv:1712.06148*, 2017.

Kim, J. and Choi, S. On local optimizers of acquisition functions in Bayesian optimization. *arXiv preprint arXiv:1901.08350*, 2019.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2015.

Lakshminarayanan, B., Pritzel, A., and Blundell, C. Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in Neural Information Processing Systems*, 2017.

Lu, X. and Van Roy, B. Ensemble sampling. *Advances in Neural Information Processing Systems*, 2017.

Mockus, J., Tiesis, V., and Zilinskas, A. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.

Oh, C., Tomczak, J. M., Gavves, E., and Welling, M. Combinatorial bayesian optimization using the graph cartesian product. *Advances in Neural Information Processing Systems*, 2019.

Paliwal, A., Gimeno, F., Nair, V., Li, Y., Lubin, M., Kohli, P., and Vinyals, O. Regal: Transfer learning for fast optimization of computation graphs. *arXiv preprint arXiv:1905.02494*, 2019.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search, 2019.

Schuchardt, J., Golkov, V., and Cremers, D. Learning to evolve. *arXiv preprint arXiv:1905.03389*, 2019.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and De Freitas, N. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

Snoek, J., Larochelle, H., and Adams, R. P. Practical Bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems*, 2012.

Srinivas, N., Krause, A., Kakade, S. M., and Seeger, M. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

Thompson, W. R. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Wilson, J. T., Hutter, F., and Deisenroth, M. P. Maximizing acquisition functions for Bayesian optimization. *Advanced in Neural Information Processing Systems*, 2018.

Wu, Z., Kan, S. B. J., Lewis, R. D., Wittmann, B. J., and Arnold, F. H. Machine learning-assisted directed protein evolution with combinatorial libraries. *Proceedings of the National Academy of Sciences*, 116 (18):8852–8858, Apr 2019. ISSN 1091-6490. doi: 10.1073/pnas.1901979116.