Optimization 8, Neimhin Robinson Gunning, 16321701

(a)

(a) (i)

An implementation of global random search is provided in `src/global_random_search.py` in the appendix. The implementation allows specifying different upper and lower bounds for each parameter when sampling random parameter vectors.

(a) (ii)

Here we compare the performance of global random search and gradient descent with constant step size on two functions, f and g:

$$f(\boldsymbol{x}) = 3(x_1 - 5)^4 + 10(x_2 - 9)^2$$

$$g(\boldsymbol{x}) = \max(x_1 - 5, 0) + 10|x_2 - 9|$$

We can note that is *possible* for the global random search to perform wildly better than gradient descent if the first random parameter vector is the global optimum, however, the performance does tend to follow consistent patterns. It's not fair to compare performance with respect to number of iterations because the global random search algorithm can perform many more (about 100 times more in this case) iterations than gradient descent in the same wall-clock time. The performance w.r.t. iterations in the top two plots of Figure 1 seems pessimistic about global random search. We compare the wall-clock performance in the bottom two plots. For f the algorithms have roughly comparable performance. In one case global random search algorithm achieves the lowest loss. For g (which has non-smoothness and regions with constant gradients), we are able to find a step size (0.003) which causes gradient descent to consistently perform much better than global random search. For GRS in all cases here we set $l_1 = 0, u_1 = 10$ and $l_2 = 0, u_2 = 18$, which puts the global minimum (5,9), at the center of these two ranges. The question of tuning these bounds is tricky, because obviously the best values would be $l_1 = 5, u_1 = 5$ and $l_2 = 9, u_2 = 9$, but it is not fair to encode this prior knowledge when considering how the algorithm would behave for functions with less known behaviour.

We find that global random search performs about 100 times more function evaluations per second than the gradient descent algorithm. Comparing the wall-clock performance is difficult here for two reasons. For one, we need to keep track of the cost function values for each parameter estimate of gradient descent, which adds memory and compute overhead to gradient descent which would not be necessary if we were running the algorithm without diagnostics. The second reason is that global random search is almost arbitrarily parrallelizable, so we could, given enough cores, bring the global random search algorithm down to constant time, give or take some coordination. For this reason in subsequent sections we will include comparisons based on the number of function or gradient evaluations required by the algorithm.

(b) (i)

We now modify the above global random search algorithm in two ways. We call the first modification `rnd search b`.

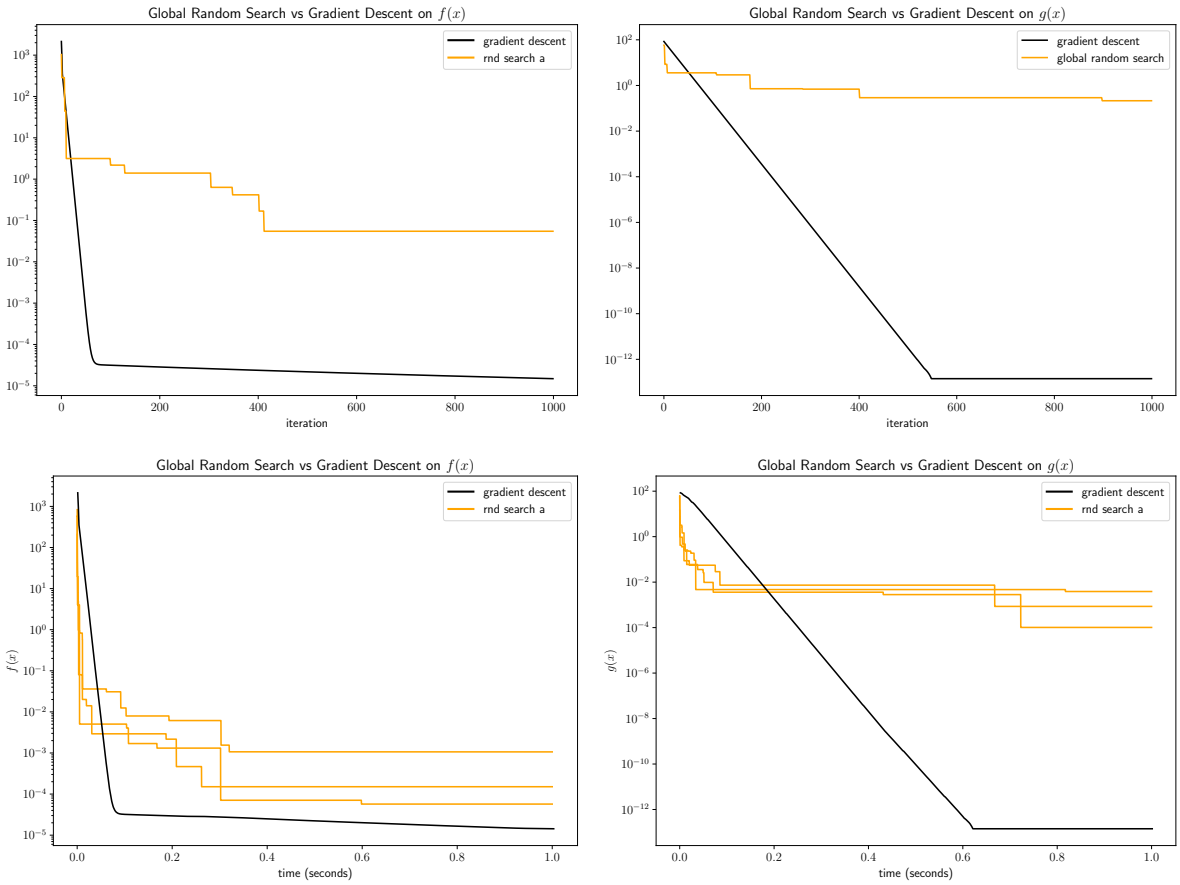

Figure 1: Global random search vs gradient descent in terms of both iterations and wall-clock time.

`rnd search b`:

Initialization Phase: Same as (a) set l_i and u_i for each parameter i. Then generate N random parameter vectors and evaluate the cost function for each. Select the M parameter vectors with the lowest cost, call these B_0.

Iteration Phase: Randomly select a parameter vector p from the M selected parameter vectors. Generate a new parameter vector by perturbing each p_i by $\delta \sim \mathcal{U}(-\alpha(u_i - l_i), \alpha(u_i - l_i))$, where α is a new hyperparameter we have to set. Repeat until N paramater vectors have been generated. Evaluate the cost function at each of the N paramater vectors and select b_i the M best of these. Join the new best M, b_i, with the old best M, B_{i-1} and select the best M of $b_i \cup B_{i-1}$ to get B_i. Repeat the iteration phase.

`rnd search b_mod`:

We call the second modified algorithm `rnd search b_mod`.

Initialization: Same as (a) set l_i and u_i for each parameter i manully.

Iteration Phase: Then generate N random parameter vectors and evaluate the cost function for each. Select the M parameter vectors with the lowest cost. Now we update all l_i and u_i based on the M selected parameter vectors, $x^1, x^2, ..., x^M$. We set $l_i := \min(x_i^1, x_i^2, ..., x_i^M)$ and $u_i := \max(x_i^1, x_i^2, ..., x_i^M)$ for each i.

We compare the performance of the three algorithms `a`, `b`, and `b_mod` in Figure 2.

(b) (ii)

Here we compare the performance of the three described random search algorithms. The top two plots of Figure 2 compare cost as a function of wall-clock time. We use $l_1 = 0, u_1 = 10$ and $l_2 = 0, u_2 = 18$ in all cases. The gradient descent step size is tuned to each function and starts at $x = [0, 0]$. For `rnd search b` we use $N = 400$, $M = 100$ and $\alpha = 0.001$. For `rnd search b_mod` we use $N = 100$ and $M = 10$.

For both functions f and g, the `rnd search b_mod` algorithm is by far the fastest to converge, and tends to actually find the global minimum, $x = [5, 9]$. However, it is possible for the algorithm to diverge, which is seen in some cases in the bottom left plot of Figure 2. This occurs when the min and max for each parameter value l_i and u_i defines an interval that does not include the global optimum. The choices of N and M are important in terms of probability that `rnd search b_mod` will diverge. If N is too low then there is a higher probability of randomly choosing paramaters 'on one side' of the global minimum, which will cause the new min and max values to lie outside the global minimum. A similar argument applies to the size of M. With low N and M we have a chance of converging much faster, but also a higher chance of diverging. With large N and M we take a safer but probably slower route.

The `rnd search a` and `b` algorithms have more comparable performance, but by tuning α (I won't discuss the details of tuning α) we have found a configuration in which `rnd search b` usually performs a little bit better than `a` on both functions f and g, given sufficient iterations. The initial convergence speed of `b` is sometimes quite poor because it depends heavily on the initial sample of parameter vectors. However, the `b` algorithm has the property that it can continue to make slow and steady progress, even if it ends up in a state where B_i consists only of bad parameter vectors. An example of this can be seen in the second row, first column of Figure 3. After a few iterations all subsequent samples from the parameter space are limited to a small region defined by B_i, which is far from the optimum.

The `a` algorithm is plauged by diminishing returns, whenever it makes progress the chances of further progress are diminished.

In Figure 3 we plot the random guesses of a run of each algorithm superimposed on each function, with $N = 100$, $M = 50$ for `b` and `b_mod`, and $\alpha = 0.01$. For `a` we use $N = 300$.

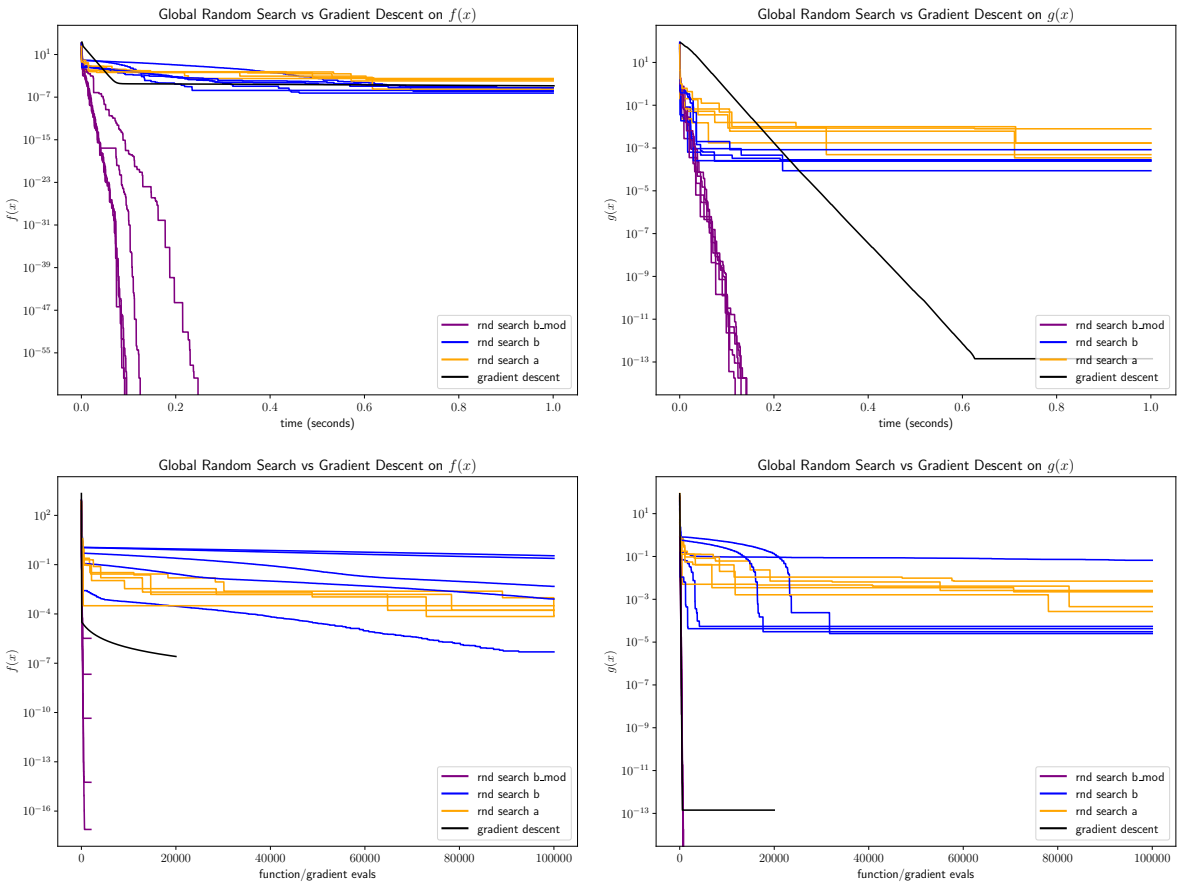

Figure 2: Comparison of the three proposed random search algorithms, top: time vs cost, bottom: evals vs cost.

. The left column has f the right colum has g. We represent time with the hue of the points, lighter (whiter) being earlier and darker (bluer) being later. We plot every subsequent sample, regardless of whether it yielded the lowest cost seen in the run thus far. In the top row (`a`) we see that the samples are independent of time, resembling the bivariate uniform. The behaviour of `b` in the second row starts out somewhat uniform (white), then samples become concentrated to a handful of disjoint regions (light blue), until finally all samples are being taken from one small region (dark blue). The third row (`b_mod`) shows that early samples are uniform and from a large rectangle, then a smaller rectangle, until eventually all samples are from a very thin rectangle encompassing the global minimum.

(c)

Here we use all three algorithms to tune the hyperparameters of CNN, which we are training in a 10 class classification problem. The hyperparameters are (i) minibatch size, (ii) adam optimizer params (α, β_1, β_2), and (iii) number of epochs, i.e. there are 5 hyper parameters. Our cost function $c(x)$ takes a parameter vector $x \in \mathbb{R}^5$,

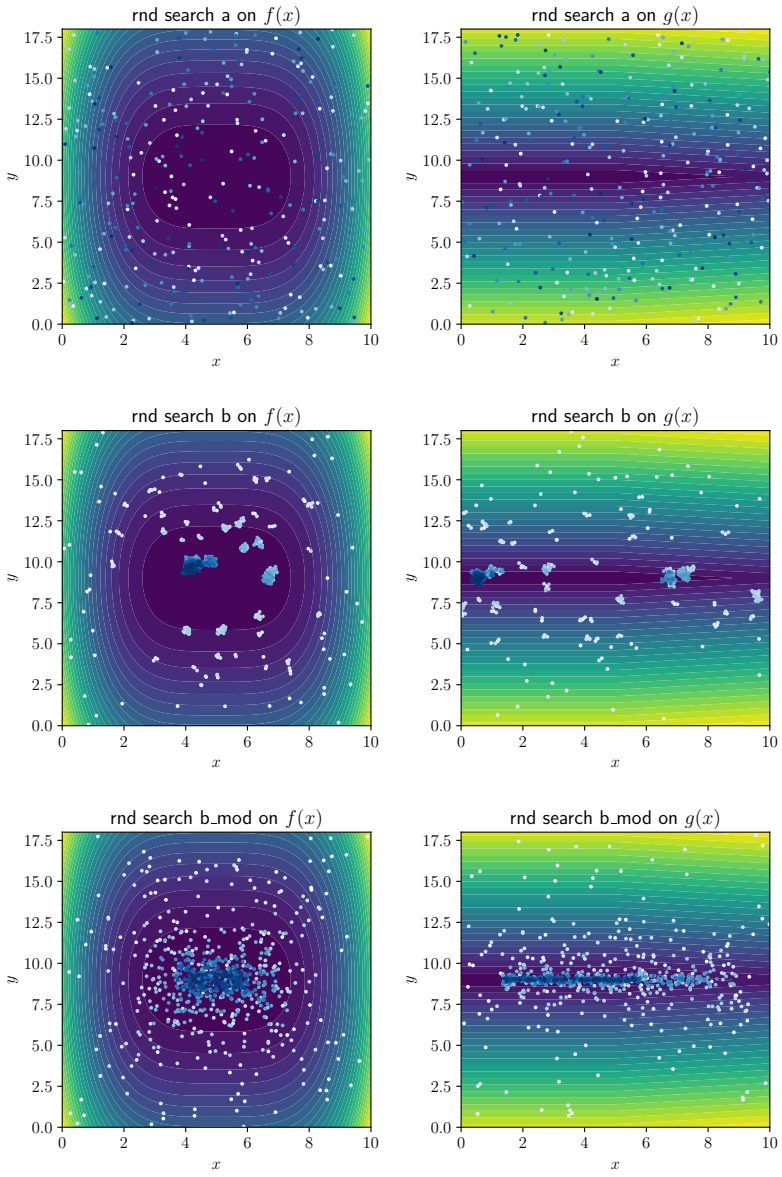

Figure 3: Progressive guesses made by each of the algorithms, superimposed onto the contour plots of the functions f and g. White dots are earlier guesses, darker blue dots are later guesses.

```
src/cps.py        Tue Apr 09 22:43:27 2024        1
    1: ps = [
    2:          {"min": 1,                "max": 1024},          # minibatch
    3:          {"min": 0.0000000001,     "max": 0.01},             # alpha
    4:          {"min": 0.5,              "max": 1},             # beta1
    5:          {"min": 0.5,              "max": 1},             # beta2
    6:          {"min": 10,               "max": 100},           # epochs
    7: ]
```

Figure 4: Min and max parameter values to initialize each random search algorithm.

	a	b	b_mod
held-out loss	1.998	1.8337	1.790

Table 1: Ultimate categorical cross-entropy losses on the held-out test set ($n = 49,000$) for each random search alg.

where the x_is are the respective hyperparameters. The cost function trains a CNN with those hyperparameters, on $n = 1000$ examples from the training set, and then evaluates the model on $10,000$ examples from the test set, returning the categorical cross entropy loss of the model on the test set. The min and max for each parameter are presented in Figure 4. It is an expensive function, especially when the number of epochs is high. As a baseline we train the model with `minibatch_size`=128, $\alpha = 0.001$,$\beta_1 = 0.9, \beta_2 = 0.999$ and `num_epochs=40`, which achieves a loss of 1.8646.

Each algorithm is given 99 function evaluations (for `a` $N = 99$; for `b` and `b_mod` $N = 33, M = 10$ and we run three iterations).

We present the value of the cost function over the number of function evaluations for each algorithm in Figure 5. Alg `b` both descends most quickly and ends up with the lowest final cost. Alg `a` starts out more promising than `b_mod` but ultimately `b_mod` has a lower final cost than `a`.

The final parameter values selected by each algorithm are presented in Figure 6. We now train 3 models with the parameters presented in Figure 6, using the same 1000 training samples, but evaluating on the categorical cross-entropy on a held test set of $49,000$ samples.

While `b` was fastest to reduce the loss and reached the lowest ultimate training loss, we find that `b_mod` has the best ultimate performance on the held-out test set (see Table 1). We suspect the reason `b` didn't perform as well on the held-out test set, is because of that algorithms ability to continue exploring indefinitely, likely leading to overfitting the hyperparameters. What happens if we stop `b` early instead? Using the parameters from `b` after 5 function evaluations we get a better loss on the held-out test set of 1.8038, but still not as good as `b_mod`. These results are tentative because we have only run each algorithm once. However, an explanation as to why `b_mod` might be ultimately better is that it converges monotonically to smaller parameter spaces, preventing excessive exploration and therefore preventing overfitting.

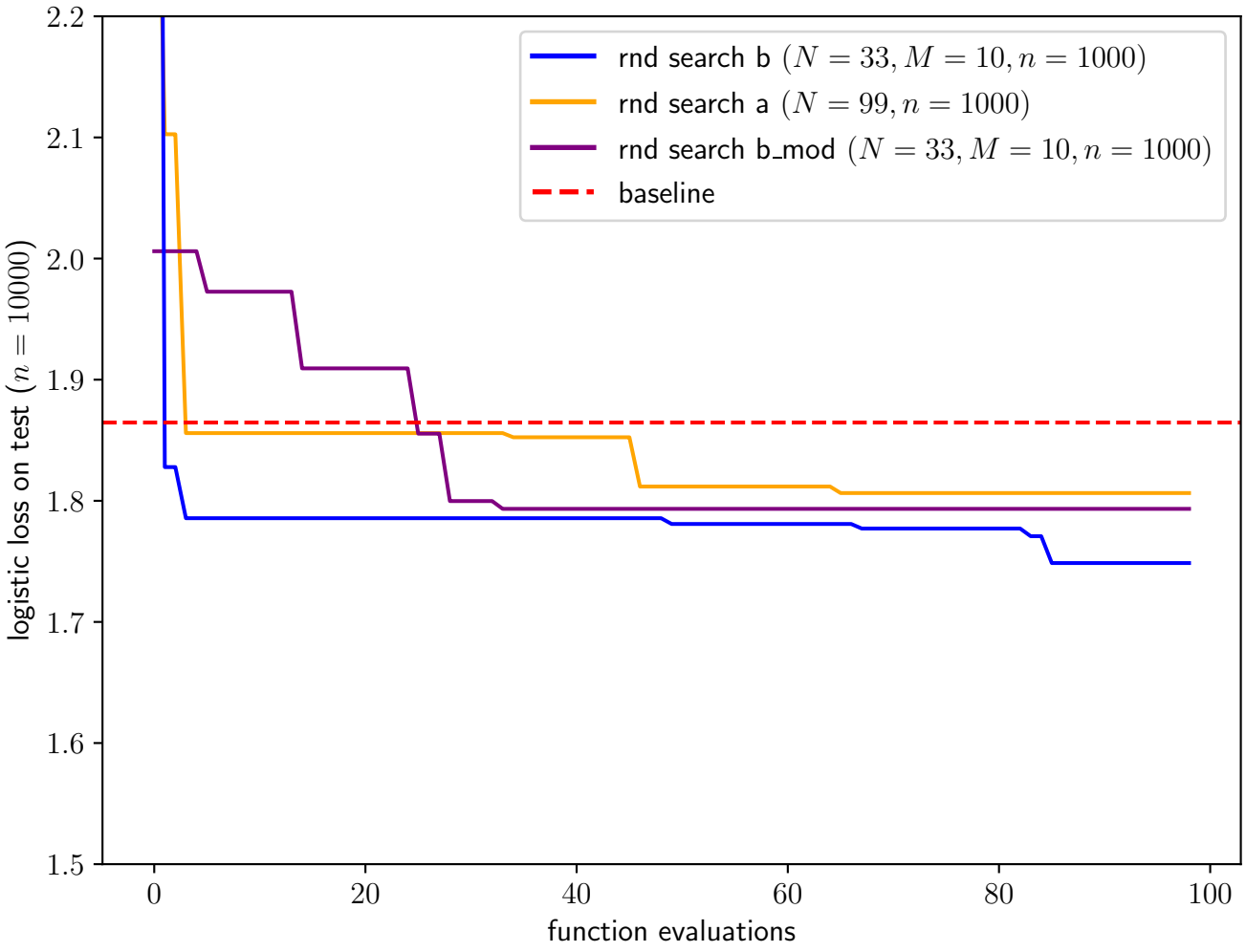

Figure 5: Training cost vs function evaluations for the CNN using three algs.

a-N99.json **Wed Apr 10 23:20:56 2024** **1**

```
{
  "best_params": [
    913.957430854217,       # minibatch
    0.0015701252586464568, # alpha
    0.6575874719325618,     # beta_1
    0.932720394784433,      # beta_2
    81.32088463431727       # num_epochs
  ],
  "best_cost": 1.8064099550247192
}
```

b-N33-M10-it3.json **Wed Apr 10 23:21:23 2024** **1**

```
{
  "best_params": [
    534.4469442210992,      # minibatch
    0.0006231460669478447,  # alpha
    0.7991814790199026,     # beta_1
    0.9007039736299371,     # beta_2
    44.05592177501114       # num_epochs
  ],
  "best_cost": 1.7486121654510498
}
```

b_mod-N33-M10-it3.json **Wed Apr 10 23:21:20 2024** **1**

```
{
  "best_params": [
    742.2428227795274,      # minibatch
    0.0009079703308546692,  # alpha
    0.8199336231638713,     # beta_1
    0.6038924210437369,     # beta_2
    64.06011278706069       # num_epochs
  ],
  "best_cost": 1.7933474779129028
}
```

Figure 6: Best params, and best training cost for each alg on the CNN task.