Hyper-Parameter Optimization

Karanpreet Benipal

Introduction

This project takes a look at some current Hyper-parameter Optimization techniques, presents the algorithms and my own R implementations and results. I started out wanting to do something with boosting, and then began to wonder if there was a better way to pick hyper-parameters than using Grid Search. I then fell into a rabbit hole, and here we are!
Finding the best hyper-parameter configuration is honestly not that important usually. But! It is fun to think about. A (possibly) high dimensional non-convex optimization problem without a gradient is definitely a challenge. Research has become more relevant and prevalent in the last decade due to Neural Networks. The neural network hyper-parameters are more significant, where as with typical bagging and boosting methods default parameters are usually pretty good, with small improvements with adjusting the hyper-parameters. However, the most important thing is, again, that it is fun to think about and there are many interesting solutions!

Optimizers

I tested 5 different Hyper-parameter Optimization techniques. In this section I will present a description of each along with psuedocode and strengths and weaknesses.

and a cleaned version of my R code. The R code is not the same as the one I ran, but rather has some extras (such as runtime calculations and test errors) removed, as well as some other particular specific to my implementation. The full R script I used is available on my Github here (hopefully I remembered to link it and to delete this).

Grid Search

Grid search involves manually constructing a grid over the search space, and evaluating each point to find the best one.

Algorithm 1 Grid Search

procedure GridSearch(grid_points, $\mathcal{D}_{\text{train}}$)
    $T \leftarrow$ expand.grid(grid_points)
    for $i \in \{1, \ldots, \text{length}(T)\}$ **do**
        $L_i \leftarrow$ objective_function($\mathcal{D}_{\text{train}}$, T_i)
    end for
    return Parameter configuration with the lowest error.
end procedure

Considerations

Strengths:
- simple
Weaknesses:
- Tests many 'bad' configurations
- Does not effectively search the space, good configurations hiding in the spaces between the points
- Requires prior knowledge - in which case manual search may be better
Extensions:
- Maybe just don't use this. Consider manual search instead of this, or use one of the other optimizers.

Here is an example of what the parameter grid points were for Random Forest:

```
rf_params <- list( # 6 x 6, so a total of 36 models were checked
  m = c(0.5, 0.6, 0.7, 0.8, 0.9, 1),
  depth = 1:6
)
```

R code: Click Me!

Random Search

Random search involves generating n points in the search space by drawing from a distribution or distributions. In my implementation each hyper-parameter has its' own distribution and the each parameter is drawn independently of the others. A main benefit is that less time is wasted on unimportant parameters. Consider this image:

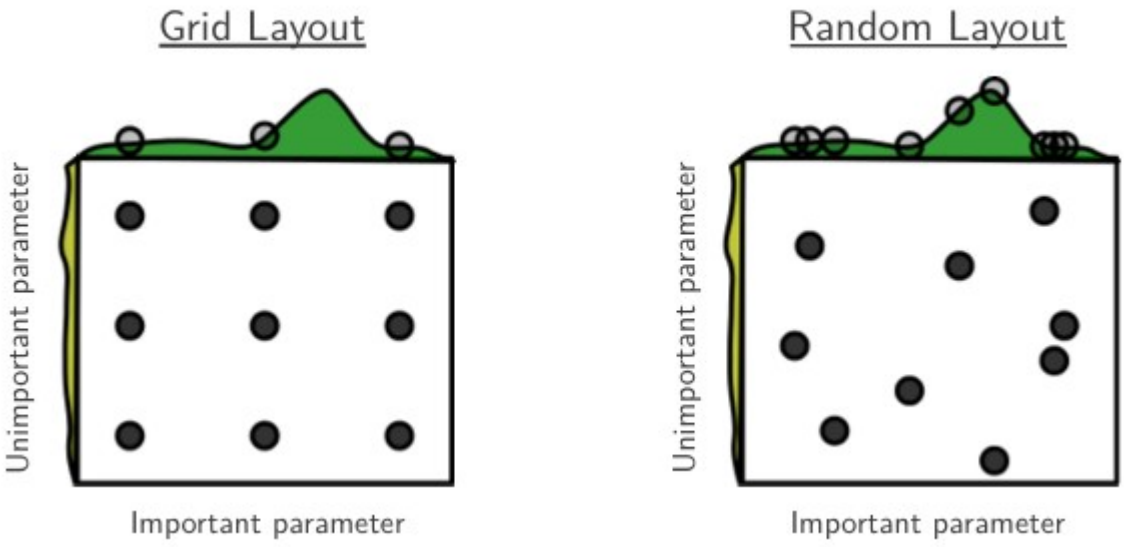


Figure 1: At the top and left are the marginal distributions. Random Search understands the important parameter far better than Grid Search, while not wasting time and resources on the unimportant parameter. Source: Figure 1 from "Random Search for Hyper-Parameter Optimmization" by Bergstra & Bengio

Algorithm 2 Random Search

procedure RANDOMSEARCH(n, dists, $\mathcal{D}_{\text{train}}$)
    $T \leftarrow$ draw.points(n, dists)
    for $i \in \{1, \ldots, n\}$ **do**
        $L_i \leftarrow$ objective_function($\mathcal{D}_{\text{train}}$, T_i)
    end for
    return Parameter configuration with the lowest error.
end procedure

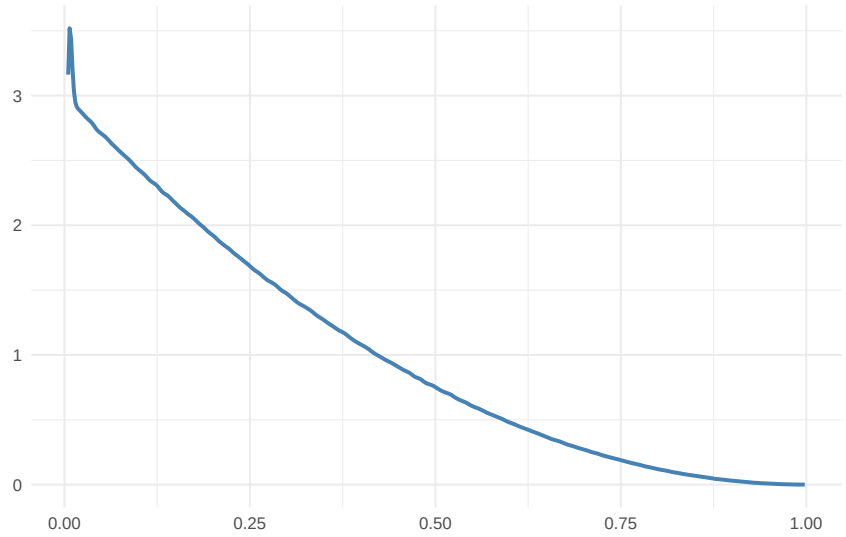
Considerations

Strengths:
- Not constrained to a grid and so can check any point
- Can set different distributions so "bad" configs are tested less often
Weakness:
- Random, so a low number of iterations may not find a decent model as space would not be covered effectively
- May need many iterations to find a close-to-best model, especially in higher dimensions
- May check multiple configs that are very similar, while some parts of the parameter space remain unexplored

Here is an example of what was used for XGBoost, note that every element in the list is a function of n which provides n draws from the corresponding hyper-parameter distribution.

```
xgb_dists <- list(
  # eta is min of 3 uniforms. favours smaller values, does not allow smaller than 0.005
  eta = function(n) pmax(pmin(runif(n), runif(n), runif(n)), 0.005),
  depth = function(n) sample(1:6, n, TRUE),  # Integer between 1 and 6 (inclusive)
  subsample = function(n) runif(n, 0.7, 1), # Uniform between 0.7 and 1
  colsample = function(n) runif(n, 0.7, 1) # Uniform between 0.7 and 1
)
```

Here's a look at the distribution for η:



R code: Click Me!

Bayesian Optimization using Guassian Processes

Bayesian Optimization is an entire topic that could fill books (and it does). In brief, the idea is to select the next point of interest based on the current points we have evaluated. Now that we have new information we select the most interesting point again. We iterate on this until we reach some stopping criteria. In my implementation, the search would stop after: a) convergence, b) a maximum number of iterations is reached, or c) the time limit expired.

Consider this image:

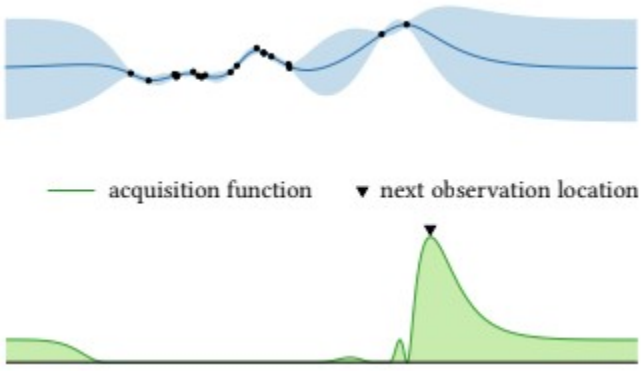


Figure 2: Guassian Process confidence intervals with acquisition function. Source: Figure 4.12 from Roman Garnett's book "Bayesian Optimization".

The idea is to first evaluate some initial points and then to fit some surrogate model to those points. In the image above (and my implementation) Gaussian Process Regression is used. Then, some acquisition function is applied. Whichever point maximizes this acquisition function is the next point we are going to evaluate. Then simply repeat the process. Since we are minimizing, I have used the negative of the objective function here, since the base algorithm is a maximizer, but you could adjust the algorithm to minimize just as easily.

Considerations

Strengths:
- Learns from previous iterations to (often after sufficient iterations) find better and better configurations
Weaknesses:
- Fitting the GP and maximizing it can take a long time, especially in higher dimensions.
- Maximizing the GP can fail to converge
Extensions:
- Consider using a grid or an Uniform Design (more below with SeqUD) as the initial grid. This will explore the space and then we can use the costly Bayes optimization to fine tune. This way we will need fewer iterations if we just start with a good understand of the objective function's surface.
- Instead of GP, one can use really any model for the response surface. Random Forest and Kernel density estimates are also common choices. The Kernel does suffer from the curse of dimensionality in higher dimensions, but Random Forest allegedly works very well, much better than GP in >20 dimensions.

R code: Click Me!

Algorithm 3 Bayesian Optimization

procedure BAYESOPT(bounds, n, t_{max}, $\mathcal{D}_{\text{train}}$)
    $n\_initial \leftarrow d + 1$, for a d-dimensional search
    $T \leftarrow uniform\_points(bounds, n\_initial)$
    for $i \in \{1, \ldots, n_initial\}$ **do**
        $T.error_i \leftarrow$ objective_function($\mathcal{D}_{\text{train}}$, T_i)
    end for
    Begin Bayesian Optimization
    for $j \in \{n\_initial + 1, \ldots, n\}$ **do**
        $fit \leftarrow GuassianProcessRegression(T)$
        $x_{next} \leftarrow ArgMax(AcquisitionFunction(fit))$
        $x_{next}.error_i \leftarrow$ objective_function($\mathcal{D}_{\text{train}}$, x_{next})
        $T \leftarrow$ append(x_{next}, T)
        if SpottingCriteriaMet() **then**
            exit
        end if
    end for
    return Parameter configuration with the lowest error.
end procedure

Hyperband

Hyperband involves strategically allocating resources to search more parameter combinations compared to random search in the same time. The paper suggests a few different ideas for what the resource should be. They are:
- Training time
- Dataset subsample
- Feature subsample

Different types of data would be more conducive to different resources. In my implementation I used dataset subsample as that works with the different datasets and objective functions I used in my tests.

First, first we need to talk about Successive Halving. Successive Halving is similar to random search, it generates some random points but evaluates them with a limited resource allocation. Then it chooses the best models and then increases the resource allocation and refits with the chosen points. The number of points is reduced by a factor of η each round.
Hyperband runs several brackets of Successive Halving with different starting resource allocations, so some brackets explore many different configurations and some spend more resources 'exploiting' a few configurations.

Considerations:
Strengths:
- Considers more configurations than Random Search in the same amount of time.

Weaknesses:
- We cannot control how resources are allocated, only how many there are. At least in the base algorithm. It seems rather arbitrary, the paper does not explain why.
- If the results of a low resource evaluation has low variance, then we do not need to repeatedly test that config multiple times. If it has high variance, then good configs may be discarded early.

Extensions:
- Try a more flexible approach, specifying the brackets manually may be preferred.
- For your specific dataset and model test if low resources still result in accurate assessments and then use a higher η, or specify brackets manually to check more configurations.

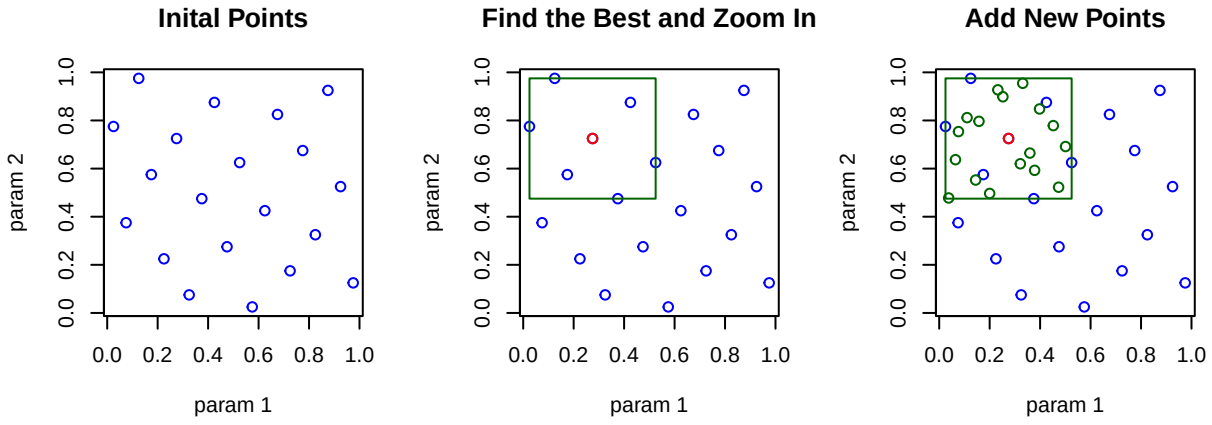
Algorithm 4 Hyperband

procedure HYPERBAND(R, η, dists, $\mathcal{D}_{\text{train}}$)
  $S_{max} \leftarrow \lfloor \log_\eta(R) \rfloor$
  $B \leftarrow (S_{max}+1)\dot{R}$
  for $s \in \{S_{max}, \ldots, 0\}$ **do**
    $n \leftarrow \lceil \frac{B\eta^s}{R\cdot(s+1)} \rceil$
    $r \leftarrow R\eta^{-s}$
    $T \leftarrow$ draw.points(n, dists)
    for $i \in \{0, \ldots, s\}$ **do**
      $n_i \leftarrow \lfloor n\eta^{-i} \rfloor$
      $r_i \leftarrow r\eta^i$
      for $j \in \{1, \ldots, n\}$ **do**
        $L_j \leftarrow$ objective_function($\mathcal{D}_{\text{train}}$, T_j, r_j)
      end for
      $T \leftarrow top\_k(T, L, \lfloor \frac{n_i}{\eta} \rfloor)$
    end for
  end for
  return Parameter configuration with the lowest error.
end procedure

R code: Click Me!

Sequential Uniform Designs (SeqUD)

SeqUD can be thought of as a "batch-sequential" algorithm. Unlike Bayes where one config at a time is selected, here we select many configs within a smaller and smaller search space. The first step is selecting points in the search space that fill the full parameter space as evenly as possible. This is determined by a discrepancy measure, my implementation uses the 'wrap-around discrepancy'. These points are created in the $[0,1]^d$ hypercube and then mapped to the hyperparameter space. Each point is evaluated, then the search space is halved along each dimension and centered at the best point. New points are added to this reduced space, again spread evenly taking the existing points into consideration as well. This is then repeated for a number of iterations. To make things clearer, consider the following plots which show as example of this in two dimensions.



Now that we have that image of the algorithm in our minds, here is the pseudo code.

Algorithm 5 Sequential Uniform Designs

procedure SEQUD(n_points, T_{max}, bounds, $\mathcal{D}_{\text{train}}$)
  $r_b \leftarrow 0.5$
  $U \leftarrow$ empty set
  for $t \in \{1, \ldots, T_{max}\}$ **do**
    if $t = 1$ **then**
      $T \leftarrow create\_UD(n\_points)$
    else
      $r_b \leftarrow \frac{r_b}{2}$
      $lwr \leftarrow best - r_b$
      $upr \leftarrow best + r_b$
      $box\_points \leftarrow \{U : lwr < U_i < upr\}$
      $T \leftarrow augmentUD(n\_points, box\_points, lwr, upr)$
    end if
    for $i \in \{1, \ldots, n\}$ **do**
      $L_i \leftarrow$ objective_function($\mathcal{D}_{\text{train}}$, T_i)
    end for
    $best \leftarrow T_{which.min(L)}$
  end for
  return Parameter configuration with the lowest error.
end procedure

Considerations

Strengths:
- Maximizes exploration of the search space for a given number of configs. Avoids checking the same or very similar configs twice.
- By searching around the previous best it act similar to Bayes search, making it less likely to miss good configs in spaces between configs like the non-sequential searchers.
- Can still be run in parallel for config evaluations, unlike Bayes.
Weaknesses:
- If the objective function has two 'peaks' only one may be explored in the subsequent iteration, which may result in the best model being missed.
- Needs some intuition to use: how to balance iterations vs number of points per iteration.
- Augmenting the UD can take some time, but usually not as much as Bayes.
Extensions:
- Instead of zooming in to just one point we could zoom in on multiple if the second best configuration is not in the new box. We could also use an adaptive size for the new box, removing the 'bad' parts of the search space first.
- As with all models we could also experiment with limited resource allocation to make it run faster.
- If augmentation takes longer than function evaluation, then just don't augment, but use the original design shrunk down to the size of the new box. This may result in similar models being checked twice but it will still be faster. Additionally we could also just remove similar configuration if they occur, in which case it will always be faster.
- Use the same distriubtions as Random Search. Every distribution function can be set to take a real number between [0,1] and use the inverse CDF to convert into a draw from that distribution. So we can convert from the $[0, 1]^d$ hypercube to the (possiblitly joint) multivariate distribution.

R code: Click Me!

Algorithm Testing

Experiment Setup

In my experiment I test four different objective functions, when given a hyperparameter configurations they return a corresponding validation error. The different models and corresponding search space dimensions are:

Model	Number of Parameters
Random Forest	2
XGBoost	4
Feed Forward Neural Network	10, 13
Convolutional Neural Network	16

This is a very limited experiment as I lack the computational resources to test these methods properly (It would takes months to run on my computer). As such we cannot draw any conclusive results from this. However, this is a decent starting place to identify some strengths and weaknesses of each one, and possibly some extensions that may help in a practical application.
The datasets I used were taken from the UCI Machine Learning Repository. Each model, except for CNN was tested on the following five datasets:

- Heart Disease [https://archive.ics.uci.edu/dataset/45/heart+disease]
- Bank Marketing [https://archive.ics.uci.edu/dataset/222/bank+marketing]
- Forest Fire [https://archive.ics.uci.edu/dataset/162/forest+fires]
- Bike Sharing [https://archive.ics.uci.edu/dataset/275/bike+sharing+dataset]
- Superconductivity [https://archive.ics.uci.edu/dataset/464/superconductivty+data]

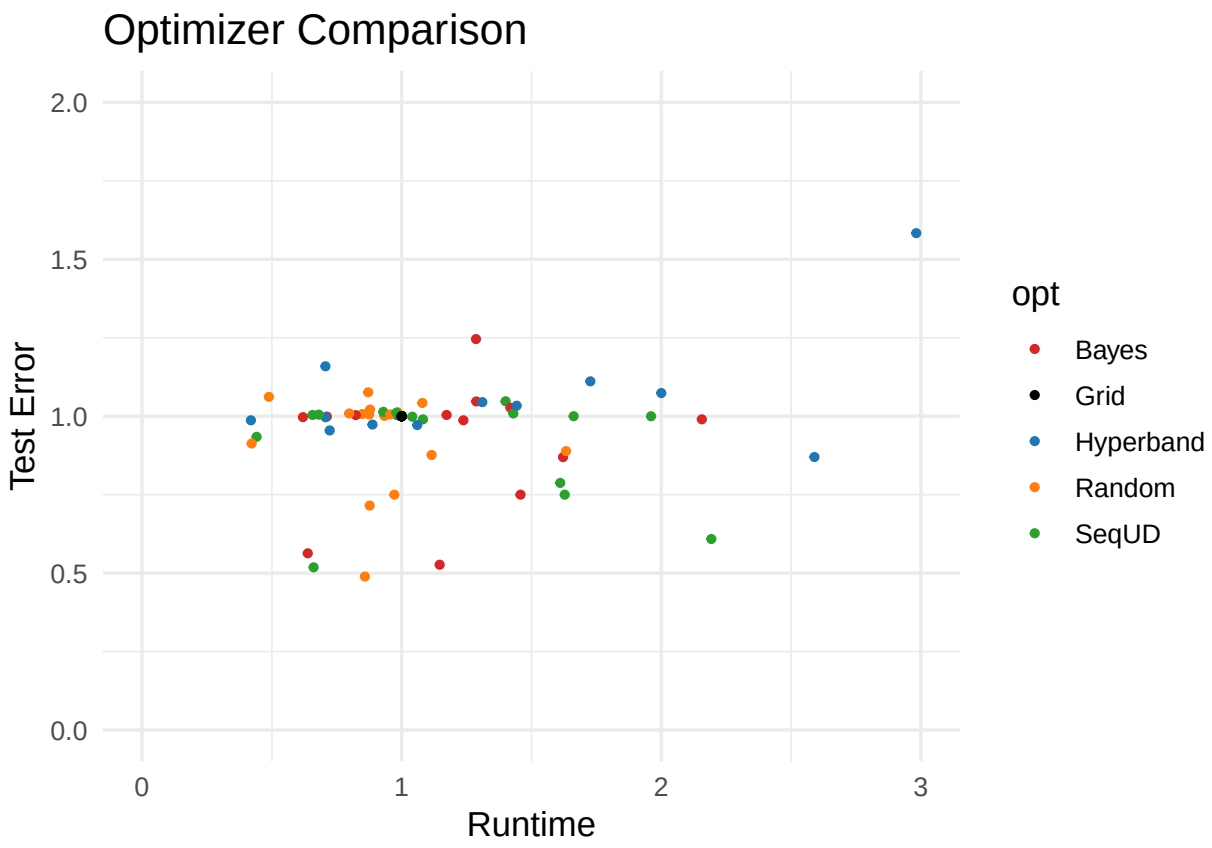
CNN was tested only on MNIST, again due to computational limitations.
Models are selected by comparing the validation error defined as follows for each model:
Random Forest: Out-of-Bag (OOB) Error
XGBoost: 3-fold Cross Validation
Neural Network: Train / Valid split
Each optimizer will return the validation and test errors of the selected configuration, as well as the runtime. The runtime includes the full run of the algorithm up-to and including the selection of the best configuration, it excludes the calculation of the test error.
The resources for the algorithms were assigned as follows:
Random Search: The same number of configurations as Grid Search.
Bayes Search: Time limit 10% more time than Grid Search for the same model and dataset, some exceptions in the case that Grid Search ran too fast.
Hyperband: Roughly set budget so that it takes a similar time to Grid Search.
SeqUD: Same as Hyperband.

Preliminary Results

There are 16 runs per optimizier for a total of 80 observations. As stated before there are not enough observations to draw conclusions. Furthermore, the best optimzier changes at different dimensions, for

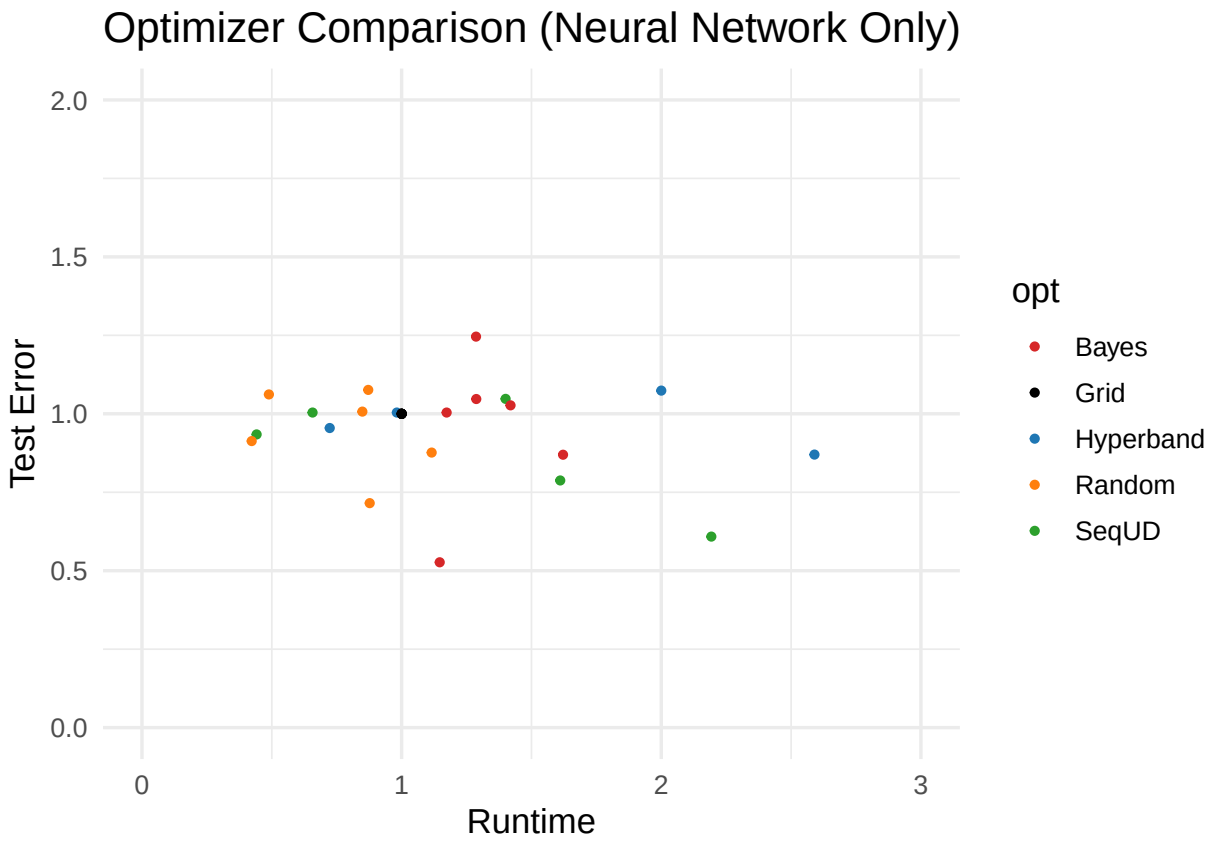
example if the objective function can be evaluatied quickly then Bayes and SeqUD are at a disadvantage since they spend time picking the next point. So we would need many obervations for each to fully understand which to choose in each situation.
Of course the runtime and errors are in very different scales for each model and dataset. I normalized the runtimes and test errors by dividing by the runtime and test error of Grid Search for that particular dataset and model.

The relative runtimes and test errors are in the plot below:



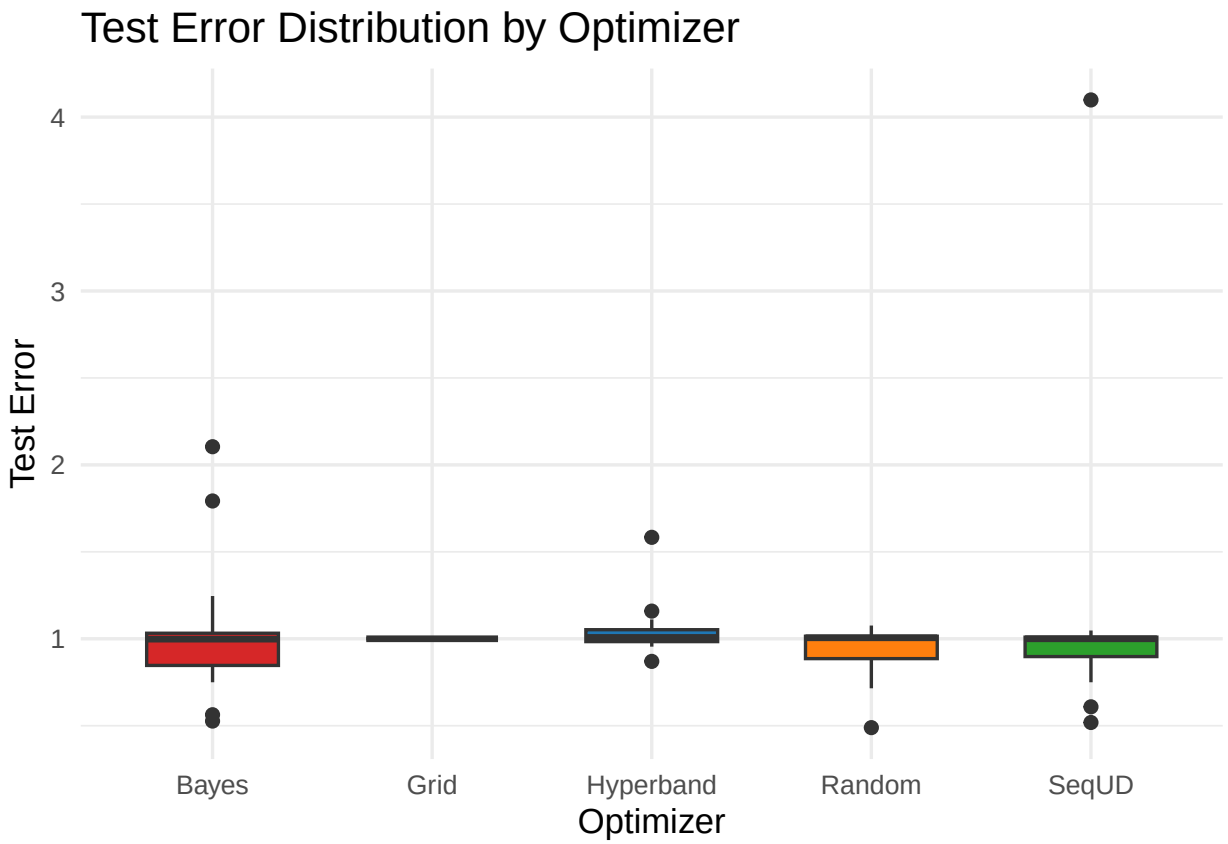
One thing that we can see is that Random Search seems to run faster than Grid Search. I think this is becuase each model has a parameter that controls how long the objective function takes to run, for example tree depth or learning rate. On average, Random Search distributions chose faster configurations than Grid Search. So it is ofcourse dependent on the grid, as we would expect Random Search to have the same runtime. It may be that making the Grid takes longer than randomly drawing, however I do not beleive that is a noticable difference. We do not see any clear patterns with the other optimizers. Let's take a look at just the neural network runs. As generally HPO is used just for neural networks and we may be able to spot patterns just looking at the higher dimensional problems.

```
## Warning: Removed 3 rows containing missing values or values outside the scale range
## (`geom_point()`).
```



Bayesian Optimzier takes a very long time to select the next point, for the CNN it took about 5-8 minutes to evaluate one configuration, but Bayes might take 60-120 minutes to pick the next point. So it tests compartively very few configurations, which is likely why it performs worse than the others here.

The runtimes were intentionally meant to be similar, so let's focus on the distribution test errors.



This plot gives the impression that Hyperband is not very good. However, again, we do not have enough data to draw clear results. There are a few things I have learned that I think I can say. The sequential algorithms (Bayes and SeqUD) do better in lower dimensions and when the objective function takes longer to run. The longer the objective function takes to run the smarter we want to be about what configuration to try, but as dimensionality increases these algorithms take significantly longer to pick new points. If we can pick good distributions for Random Search it works very well. Never use Grid Search. Hopefully this was interesting to you!

Appendix

R Code

Grid Search Code

Click here to go back

```
grid_search <- function(X, y, obj_func, param_values) {
  # Create the grid
  grid <- expand.grid(param_values, stringsAsFactors = FALSE)

  # Evaluate objective function at each point
  n <- nrow(grid)
  errors <- numeric(length = n)
  for(i in 1:n) {
```

```
    errors[i] <- obj_func(X, y, params=grid[i,])
  }

  # find & return best model (lowest error)
  return(grid[which.min(errors), ])
}
```

Random Search Code

Click here to go back

```
random_search <- function(X, y, obj_func, param_dists, n) {
  # get n many realizations of the parameters
  grid <- data.frame(row.names = 1:n)
  for(p in names(param_dists)) {
    grid[[p]] <- param_dists[[p]](n)
  }

  # Evaluate objective function at each point
  errors <- numeric(length = n)
  for(i in 1:n) {
    errors[i] <- obj_func(X, y, params=grid[i,])
  }

  # find & return best model (lowest error)
  return(grid[which.min(errors), ])
}
```

Bayesian Optimization Code

Click here to go back
This one is really simple since the bayesOpt function does all the work for us.

```
bayes_search <- function(X, y, obj_func, param_bounds, n, t_limit=NULL) {
  scoreFunction <- function(...){
    # negative since bayesOpt is a maximizer
    list(Score = -obj_func(X=X, y=y, params=list(...)))
  }

  init_points <- max(6, length(param_bounds)+1)

  optObj <- ParBayesianOptimization::bayesOpt(
    FUN = scoreFunction,
    bounds = param_bounds,
    initPoints = init_points, # randomly eval at some points initially
    iters.n = n,
    iters.k = 1,
    otherHalting = list(timeLimit = t_limit)
  )
  # Return Results
  return(data.frame(getBestPars(optObj)))
}
```

Hyperband Code

Click here to go back

```r
hyperband <- function(X, y, obj_func, param_dists, R, eta = 3) {
  all_results <- list()

  # maximum number of brackets
  s_max <- floor(log(R, base = eta))
  B <- (s_max + 1) * R

  for(s in s_max:0) {

    n <- ceiling((B / R) * eta^s / (s + 1))
    r <- R * eta^(-s)

    # Begin Successive Halving in inner loop
    # get n many realizations of the parameters
    grid <- data.frame(row.names = 1:n)
    for(p in names(param_dists)) {
      grid[[p]] <- param_dists[[p]](n)
    }

    for(i in 0:s) {
      n_i <- nrow(grid)
      r_i <- r * eta^i

      # Get losses
      L <- numeric(length = n_i)
      for(j in 1:n_i) {
        # r_frac is normalized resource allocation
        L[j] <- obj_func(X, y, params = grid[j, ], r_frac = r_i/R)
      }

      grid$error <- L
      all_results[[length(all_results) + 1]] <- grid

      # Update grid to be just the top n_i / eta performers
      k <- max(1, floor(n_i / eta))
      grid <- head(grid[order(grid$error), ], k)
    }
  }
  # Combine everything
  results <- do.call(rbind, all_results)
  # Find & Return best model
  return(results[which.min(results$error), ])
}
```

SeqUD Code

Click here to go back

```r
augmentUD <- function(X0, add, n_candidates = 1000) {
  # X0: existing design (rows = runs, cols = factors), scaled to [0,1]
  # add: number of new points to add
  # n_candidates: number of candidate points to try each step
  d <- ncol(X0)
  new_points <- c()

  for (k in 1:add) {
    # generate random candidate points
    C <- matrix(runif(n_candidates * d), ncol = d)
    # score each candidate by discrepancy when added
    scores <- apply(C, 1, function(cand) {
      SFDesign::uniform.crit(rbind(X0, cand))
    })
    # pick best candidate
    best <- C[which.min(scores), , drop = FALSE]
    # update design
    X0 <- rbind(X0, best)
    new_points <- rbind(new_points, best)
  }
  new_points
}

seqUD <- function(X, y, obj_func, param_bounds, T_max, n_points){
  d <- length(param_bounds)

  # Find range and min of each param
  mins <- sapply(param_bounds, function(x) as.numeric(x[1]))
  lwr <- mins
  upr <- sapply(param_bounds, function(x) as.numeric(x[2]))
  ranges <- sapply(param_bounds, function(x) as.numeric(diff(x)))
  is_int <- sapply(param_bounds, function(x) is.integer(x))

  from_01_to_ps <- function(U) {
    out <- matrix(NA, nrow = nrow(U), ncol = d)
    for (j in 1:d) {
      vals <- mins[j] + U[, j] * ranges[j]
      if (is_int[j]) {
        vals <- round(vals)
      }
      out[, j] <- vals
    }

    colnames(out) <- names(param_bounds)
    as.data.frame(out)
  }

  U01 <- c()
  grid <- c()
  box_radius <- 0.5

  for(t in 1:T_max) {
    # Generate points
```

```r
  if(length(U01) == 0){ # Make fresh design
    new_points <- uniformLHD(n = n_points, p = d)$design
    # map to parameter space
    ps_points <- from_01_to_ps(new_points)

  } else { # Augment existing
    # update boundaries
    box_radius <- box_radius * 0.5
    lwr <- best - box_radius
    upr <- best + box_radius

    # Check if the box is outside [0,1]^d
    shift_lwr <- pmax(0 - lwr, 0)   # how far below 0
    shift_upr <- pmin(1 - upr, 0)   # how far above 1 (negative)

    # total shift = whichever adjustment is needed
    shift <- shift_lwr + shift_upr

    # apply shift to corners
    lwr <- lwr + shift
    upr <- upr + shift

    # Find which points are in the new bounds
    in_the_box <- apply(U01, MARGIN=1, FUN = function(x) {2 * d == sum(c(x > lwr, x < upr )  )})
    box_points <- U01[in_the_box, , drop=FALSE]

    # Augment the sub design
    n_e <- n_points - nrow(box_points)
    new_points <- augmentUD(sweep(box_points, 2, lwr, "-") / (box_radius * 2), add = n_e, n_candidates

    # Shift the design back into place
    new_points <- (new_points * (box_radius * 2))
    new_points <- sweep(new_points, 2, lwr, "+")

    # Generate points in the parameter space
    ps_points <- from_01_to_ps(new_points)
  }

  # Evaluate each point
  n <- nrow(ps_points)
  errors <- numeric(length = n)

  for(i in 1:n) {
    errors[i] <- obj_func(X, y, params=ps_points[i, ])
  }
  # Save the errors
  ps_points$error <- errors
  # Append new points onto old points
  grid <- rbind(grid, ps_points)
  U01 <- rbind(U01, new_points)
  # Find the best point
  best_idx <- which.min(grid$error)
  best <- U01[best_idx, ]
```

```
  }
  # Return results
  return(grid[best_idx, ])
}
```

Bibliography