# Careful Steps in the Dark: Gradient-Free Parameter Update Methods for Deep Neural Networks

Matthew Resnick

December 9 2020

**Abstract**

When it comes to updating the parameters of a deep neural model, recent advances in gradient-based steps have provided a luxuriously efficient scheme. However, it is not always the case that gradients are available for updating a neural model —as in many AutoML cases—and sometimes when they are available, they do not provided the optimal parameter choices. In this project, I will explore several alternatives to traditional gradient-based updates by analyzing some existing works and presenting novel approaches in the domain of gradient-free optimization. We are still perhaps a long way off from these ideas being as effective in their own domain as gradient methods are in theirs, but it's important to note that the aim of this project is to explore the dynamics of these ideas to provide a deeper understanding of the road ahead, rather than exceeding performance benchmarks. I add to the work of Random Search Optimization, implement a new version that improves performance significantly, and present an original gradient-free update scheme that can outperform Random Search Optimization in testing benchmarks and can improve training time by as much as 20×.

## 1 Introduction

With the advent and success of stochastic gradient descent (SGD) [1] and related methods for updating the parameters of a deep neural network, there has been little cause for a search for alternative methods in the last few years. That's not to say there has been stagnation in the area of parameter optimization, but little headway has been made towards update methods that operate on entirely different principles. Perhaps this is a sensible outcome; after all, methods like SGD are fast, efficient, and highly effective at reaching optimal parameter values regardless of the size or architecture of a given model. However, I believe it is worthwhile to consider alternative approaches to update methods due to the fact that some models are not able to properly propagate gradients. For instance, meta-learning/AutoML models can potentially run into this issue when they have deep models controlling the generation of other deep models and cannot propagate information from the built model's objective function back to the builder model's parameters. Further, many gradient-based approaches are not always able to produce an optimal model in the first place, due to the fact that the gradients may not contain information about aspects like model structure or induced sparsity.

For these reasons, in this report I will create, analyze, and discuss several alternative parameter update methods which address the aforementioned issues. First, I will look at Random Search Optimization (RSO) from Tripathi and Singh [2] to compare the effectiveness of different update value sampling methods with traditional SGD. I will also introduce two new versions of the approach called Ternary and Binary Random Search Optimization, respectively, and demonstrate their functionality. Last, I will introduce a novel gradient-free update method based on Thompson Sampling, where the underlying idea is to more efficiently select parameters for update even without gradient information. For my experimentation, I largely make use of MNIST [3], the UCI Mushroom dataset [4], and the UCI MiniBooNE dataset [4], for several reasons. First, they are all fairly small datasets both by number of samples and number of features, which makes all of the experiments possible in a somewhat reasonable amount of time. Even a model which could obtain decent results on CIFAR-10 would take an enormous amount of computation time even for the fastest of methods, so this was a major concern. Additionally, MNIST is well benchmarked and allows for me to experiment with convolutional models while the Mushroom and MiniBooNE datasets allows me to experiment with dense models and will reveal the efficacy of my approaches on smaller loss movement scales and verify

the improvements made by my approaches. The results for MNIST can be found in the bulk of the write-up, while the results for UCI Mushrooms and MiniBooNE can be found in the Appendix.

## 2 Background and Related Work

There are two primary related works, each of which is central to the two main approaches of this project. I begin with Random Search Optimization from Tripathi and Singh [2]. The principle idea behind this paper is to update each parameter value in a model with essentially no prior information. A model's parameters are initialized in the usual way, for instance by sampling from the Kaiming normal distribution, and the standard deviation of each layer, $\sigma_d$, is recorded. Then, starting from the layer nearest to the output and moving backwards, an update of each parameter is considered by adding to or subtracting from it and then obtaining the loss for each via a forward pass of a minibatch. Then, if some movement of the value improved the loss, the update is kept. Otherwise, it is rejected and the value of that specific parameter is kept the same. The way the add/subtract value is determined has a few different options, but in the paper they only demonstrated sampling from $\mathcal{N}(0, \sigma_d)$. This update value is then annealed linearly so that it is only 1/10th of its original value by the last epoch of training. This approach is a surprisingly effective method for optimizing the parameters of a model without gradients, and can even do it in surprisingly few iterations. The only caveat is that each iteration can take a substantial amount of time, as a minibatch must be forward-passed through the model three times for every parameter for just one iteration.

The second work, A Tutorial on Thompson Sampling from Russo et. Al. [5], is a detailed review of the central idea from a much older paper, now called Thompson Sampling [6], that is a method for building sampling distributions for multiple options over time. This approach has largely found success in the area of reinforcement learning, as it is a method for confronting the multi-armed bandit problem. The idea is fairly simple, and it translates into a somewhat simple implementation as well. If one has multiple actions they would like to take, each of which returns some idea of "reward," but the taking of one action precludes the taking of another, then the following method of sampling actions can be performed when no prior information is available. Begin by sampling actions uniformly randomly, and record the reward provided by that action over time. Every time a new reward is received, a distribution can be built based on the statistics of the observed rewards that alters how often that action is selected as well as the confidence in that frequency of selection. The simplest distribution one could build with this method is a binomial distribution, where the sole parameter of the distribution is updated based on the success/failure ration of each action. In the case of a scalar reward value, success and failure could be determined by positive reward vs negative reward, or zero reward vs negative reward. However, it is also possible to build a number of other distributions for sampling based on the strength of such a scalar reward beyond just success/failure. My parameter update method, which will be described in detail in the Methods section, uses this idea but treats the update of a parameter in a deep model by a predetermined amount as a single action. There are some works that propose somewhat similar ideas, however they seem to be focused on either reinforcement learning tasks, such as [7] and [8], or simply determining an uncertainty measure for parameters, such as [9]. To my knowledge, my approach is the only attempt to directly update weight parameters using Thompson Sampling.

## 3 Gradient Free Updates - RSO

In their original paper on Random Search Optimization, Tripathi and Singh update each parameter value in a model with assuming no prior information, and yet a determination must be made about how big of a step to take every time an update occurs [2]. The authors suggested three methods of sampling a value: for a layer standard deviation $\sigma_d$, they suggest sampling from $\mathcal{N}(0, \sigma_d)$, sampling from $\mathcal{U}(-2\sigma_d, 2\sigma_d)$, and sampling from $\{-\sigma_d, 0, \sigma_d\}$. They demonstrated the efficacy of the normal sampling method, but only alluded to the efficacy of the other methods. In this section, I explore various methods of selecting update values for model parameters when using random search for optimization, as well as the influence of loss function choice. The results of this investigation reveal some interesting aspects of selecting update values when gradient information is not available and extends the work of [2].

### 3.1 RSO Sampling Analysis

First, I use a baseline model to see the effect, if any, of the loss function on the performance of RSO. In [2], the authors make exclusive use of Cross Entropy (CE) loss, but without gradient information this choice is somewhat arbitrary. It is not an unconventional metric for the type of task it is applied to, image classification for MNIST and CIFAR-10, but neither are several other loss functions.

The baseline model I use consists of 6, 16 channel/3x3 filter convolutional layers each followed by ReLU activation, the second of which is followed by a 2x2 pooling layer and the last of which is followed by global max pooling. The final layer is a dense layer, which is activated by softmax. I include but do not update the parameters for batch normalization layers, as described in the original paper. I then linearly annealed the sampling standard deviation for updates down to 1/10th of its original value, and I did updates in individual cycles by going back to front through the network. I then use both Cross Entropy and Mean Squared Error (MSE) loss functions for comparison on MNIST.

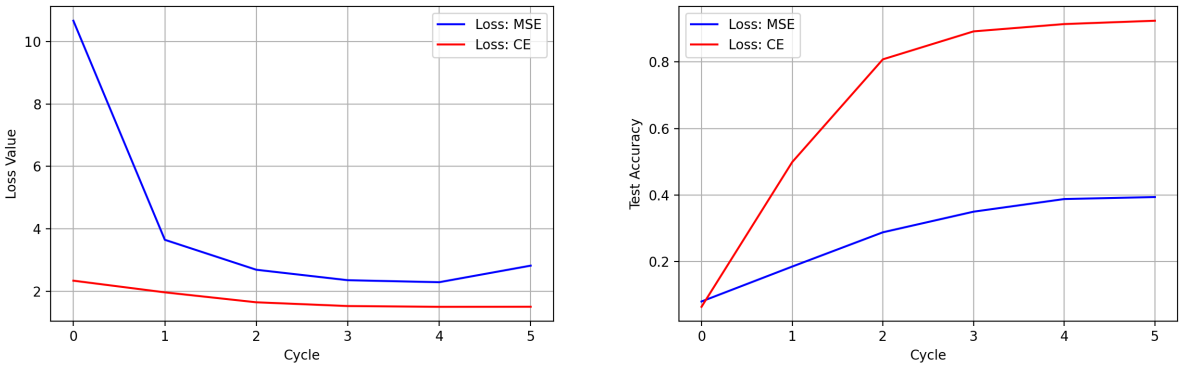


Figure 1: (Left) Loss plot for both MSE and CE versions of RSO. (Right) Test accuracy between the two versions. Both versions use normal update sampling.

Based on the extensive search process of RSO and the fact that a conventionally trained model can use MSE to optimize parameters, RSO should be able to use it just as well as CE. However, these results show that the MSE version could not even reach the first-cycle accuracy of CE even after multiple cycles of updates, and that the CE version then reached fairly decent results in just a few cycles. I believe the explanation for this is the roughness of the underlying loss surfaces, which adds an extra layer of difficulty for a model in which the loss surface cannot even be convex: how would one choose an appropriate loss function?

Next I look at the results of the three different sampling methods described previously, where $\sigma_d$ is the standard deviation of a given layer $d$, normal is sampled from $\mathcal{N}(0, \sigma_d)$, uniform is sampled from $\mathcal{U}(-2\sigma_d, 2\sigma_d)$, and fixed is sampled from $\{-\sigma_d, 0, \sigma_d\}$. The results for MNIST can be seen below.

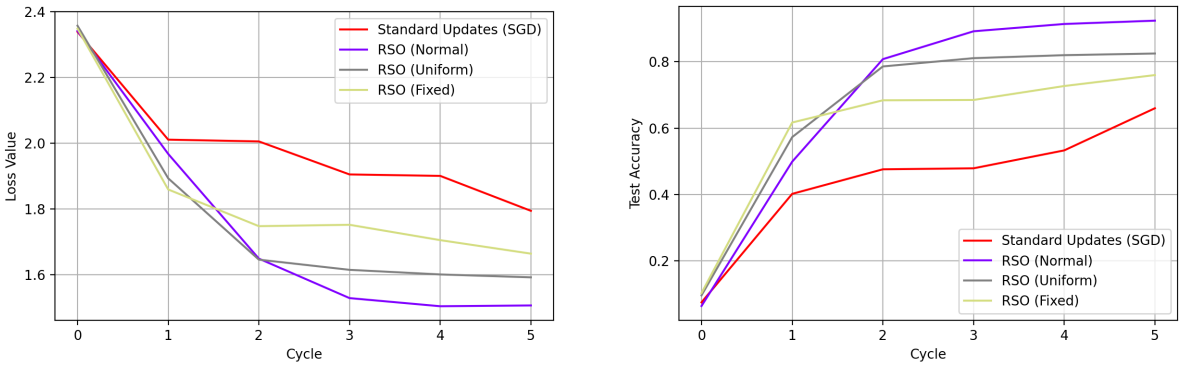


Figure 2: (Left) Loss plot for normal, uniform, and fixed update sampling methods. (Right) Test accuracy between the three sampling methods.

These results may indicate why the authors of [2] neglected to include these results in their paper, as they are not as promising as the results for the updates sampled from the Gaussian distribution. They clearly can achieve decent results, but by the fifth cycle, the Gaussian version is significantly far ahead of the others by test accuracy. At the very least, we can see each RSO method outperforming standard SGD updates in this regime, especially considering SGD saw many more samples per cycle, as RSO only sees one batch of 5,000 samples while SGD sees the entire dataset. However, it is likely that past this point is where SGD would shine anyway due to SGD having a smaller overall step size. And most importantly, these results do not show a significant factor: training time. SGD takes a total of 8.4 seconds to train 5 cycles, while each RSO method takes 23.6 minutes. For the rest of this project, my main goal is to reduce this training time and while retaining the same level of performance or better.

## 3.2 Binary RSO

It's clear from the train time difference that RSO is not practical for updating networks where gradient information is available, but when gradient information is not available it may be necessary to use something like it. Therefore, any improvements that can be made to the efficiency of RSO and similar approaches would greatly benefit endeavors into this relatively unexplored territory. There are a number of potential improvements with the existing approach, such as parallel computation and parameter correlation analysis, but I propose a less expensive, more immediate remedy.

It was demonstrated in *Binarized Neural Networks* [10] that using just two values for parameters and activations, $\{-1, 1\}$, could produce a highly effective model which is both energy efficient and computationally fast. Thus, I suspect that it may be possible to replace the search in RSO with exact values of -1, 0, and 1 (or just -1 and 1) instead of individual updates, and I show that effective networks can still be produced but at a significantly reduced computational cost. This proves to be a highly efficient approach due to the fact that real-valued weights don't even need to be stored in the first place since the backpropagation step does not occur. So I use the previously introduced baseline model with ternary weights randomly sampled from $\{-1, 0, 1\}$ as well as binary weights randomly sampled from $\{-1, 1\}$, but modify RSO so that it checks for value replacements, rather than update steps. I also add a deterministic sign function after every ReLU activation as described in [10]. I will refer to these new approaches as Ternary RSO and Binary RSO, respectively, the results of which for MNIST can be seen in the plots below.

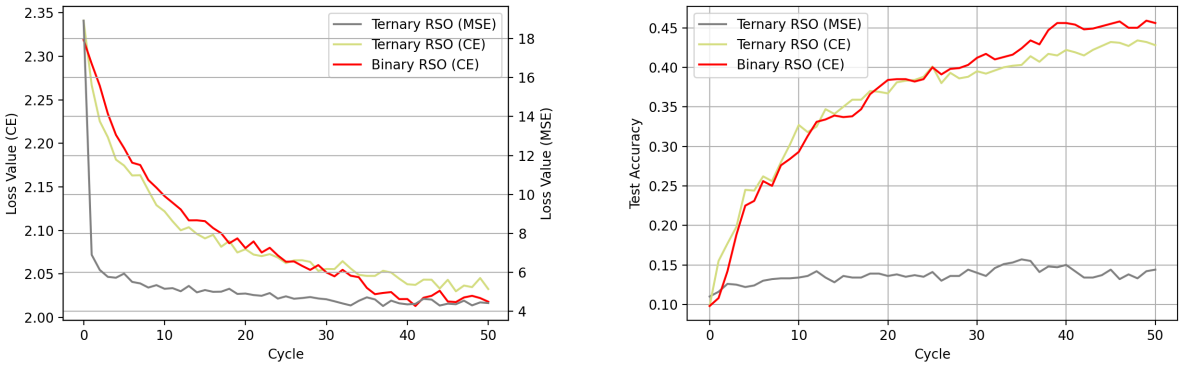


Figure 3: (Left) Loss plot for ternary (-1,0,1) and binary (-1,1) RSO updates using CE loss, alongside ternary updates using MSE loss. (Right) Test accuracy between the three versions.

A few things can be seen from these results. The first is that the MSE version converges much faster than both CE version, however the difference in test accuracy reveals the superiority of the latter two. This indicates that, as with standard RSO, the choice of objective function is of great importance. Another thing to note is that, even with an extended amount of training cycles, the best performing binary/ternary RSO updates do not come close to state-of-the-art performance. However, the additional training cycles do not translate directly to the increase in training time, due to the efficiency of the ternary and binary updates/forward passes.

| **Approach** | Standard RSO | Ternary RSO | Binary RSO |
|---|---|---|---|
| **Time per Cycle** | 23.6 min. | 20.0 min. | 9.43 min. |

Clearly, Ternary RSO is a bit faster than standard RSO only because I do not optimize the calculations during the forward pass for -1, 0, and 1, although it is possible to do so and see a large boost in speed as shown in [10]. Binary RSO also does not utilize this theoretical boost, but it is still significantly faster than the other two due to the fact that, for every update check, the network must only have one extra forward pass, rather than two.

These results are still somewhat successful, as they don't reach stellar performance but at least indicate that a fairly effective representation can be learned this way. The training itself is much faster than standard RSO —especially for the binary version—and the test accuracy is substantially better than random guessing. However, it should be possible to obtain substantially better accuracy without losing the saved training time, so I make a few improvements and show the efficacy. First, remove the layer which binarizes activations. It's not essential to this model, and having real-valued activations improves precision during a forward pass. I also combine these real-valued activations with batch-normalization layers (without learn-able parameters). The results are conclusive:

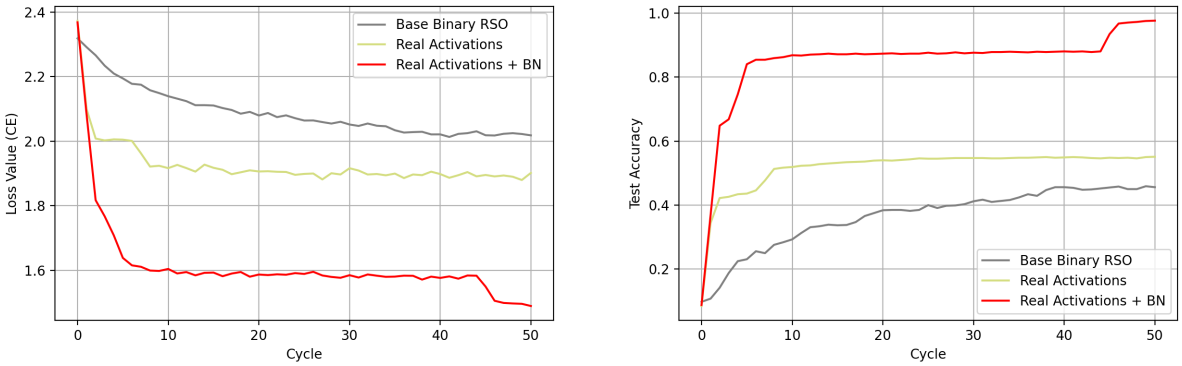


Figure 4: (Left) Loss plot for standard binary RSO, RSO with real-valued activations, and real-activated, binary RSO with batch-norm layers. (Right) Test accuracy between the three versions.

It can be seen from these results that it is possible to obtain fairly good results, even state-of-the-art, without needing the use of gradients or weights beyond -1 and 1. Additionally, all of this is faster than base RSO and reaches better test accuracy. In the next section, I will build off of these results to reach an even faster and better-performing result.

# 4 Gradient Free Updates - Thompson Sampling

After working with RSO and variants for some time, I was frustrated by the amount of computation time that was inherently needed to run it effectively. What frustrated me the most is that I believe there must be a more clever way of selecting parameters for update, even without gradient information. Thus, I turn to Thompson Sampling [6] [5], a method of confronting the multi-armed bandit problem and generally used in reinforcement learning (RL). However, instead of treating the arms as actions determined by the network, as most RL literature considers, I instead treat the parameter updates themselves as the arms. That is, I build sampling distributions for each parameter in a given network that is updated over time based on the loss previously seen by updating that parameter. To my knowledge, this is a novel approach to gradient-free parameter updates.

One important consideration is how to sample the update values. I only intend to use Thompson Sampling as a method of sampling which parameters to update and not necessarily how to update them, and thus we still face the obstacle how big of a step to take with each update. There are three conventional approaches to this. The first is the least practical, which is to choose a small step value that every parameter shares. This method is akin to gradient descent's learning rate parameter, however we don't have the same confidence in the direction of each step, and thus this approach may take a prohibitively long time to converge. The remaining two approaches are more clumsy, but hopefully faster; choose updates in the same manner as we did with RSO. That is, either sample from a pre-determined distribution for the update value, or fix the parameters to be in a set of just a few values. In this project, I only employ the latter two approaches.

In order to implement Thompson Sampling, I retained an external data structure with an array for every parameter in every layer in a given network, which I will refer to as the *belief*. What the *belief* ultimately stores depends on the type of Thompson Sampling used, but it will at least be information about sampling statistics, initialized with the most naive information about sampling each parameter as possible; every parameter is sampled uniformly randomly. If a parameter is selected, the updates discussed previously are tested via obtaining a loss from a forward pass of a minibatch with each. Then, the sampling statistics for that particular parameter are updated based on the change in loss, and if the loss went down then the base network's parameters are updated as well. Again, how the sampling statistics are updated is dependent on the type of Thompson Sampling, but the basic idea is to increase the likelihood of sampling a parameter the more it results in a decrease in loss when updates are tested on it (and, as it stops providing improvements, decrease the likelihood). In order to better facilitate this time-dependent sampling, I also test a method of forgetting as I track the statistics of each parameter, where only the recent past is considered.

## 4.1 Beta Sampling

The simpler and less computationally expensive sampling approach for Thompson Sampling is to treat the outcome of an update test as a Bernoulli process (i.e. decrease in loss is a success while no change or increase is failure), and thus we must employ the Bayesian conjugate prior of the Bernoulli distribution, the Beta distribution, to sample model parameters for update. Tracking the statistics of this approach is straightforward, as the Beta distribution's two parameters, $\alpha$ and $\beta$, can simply be represented by the number of times an update of a model parameter results in a decreased loss or not, respectively. This means that for every model parameter we also only need to keep track of two values, or one array of values, depending on implementation choices. This approach does not allow us to inform the sampling distributions with the strength of the loss decrease, but it is at least more informed than simply selecting every parameter as in RSO.

For the first tests, I look at both the base success of Beta Thompson Sampling, as well as the results of using different annealing schedules. One is the basic linear annealing, one is piecewise linear such that the first 40 cycles anneal to 1/2 of the original value and the remaining 10 anneal to 1/10, and one piecewise linear such that the first half of cycles anneal down to 1/10 of the original value and then increase back to the original value in the second half.

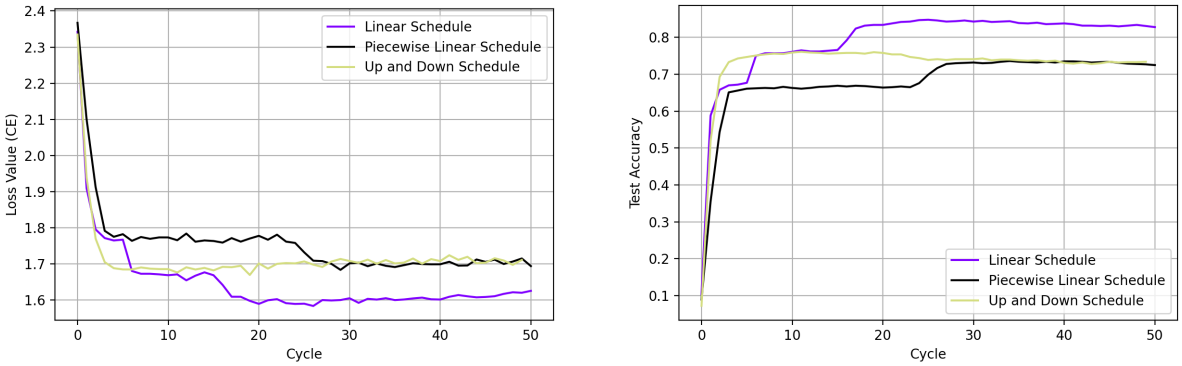


Figure 5: (Left) Loss plot for Beta Thompson Sampling with different update value annealing schedules. (Right) Test accuracy between the three versions.

So clearly, the most basic form of Thompson Sampling does work quite well, and in it also performs about 3 times faster than RSO updates. It doesn't reach quite the same level of performance, but it is possible to simply get a jump start with Thompson Sampling and then switch to RSO to do the remaining fine-grained optimization. However, there's still more to look at with Thompson Sampling alone. Another result from these tests is that linear annealing performs the best in essentially every regard, and further that the results are quite sensitive to the schedule. This indicates that finding the optimal schedule, or perhaps finding an alternative to a schedule altogether, may be an avenue for improving the results in the future.

## 4.2 Gaussian Sampling

In order to capture information about the strength of loss decrease in our sampling distributions for each model parameter, we must turn to Gaussian sampling. However, using Gaussian sampling is a bit more complicated to implement, requires that we track more values per model parameter, and is more computationally expensive overall. Since we don't have the true mean or variance of the results, we must use the normal-gamma conjugate prior and keep track of parameters $\mu_0$, $\kappa$, $\alpha$, and $\beta$ and update them via:

$$\mu_0 \leftarrow \frac{\kappa\mu_0 + n\bar{x}}{\kappa + n}$$
$$\kappa \leftarrow \kappa + n$$
$$\alpha \leftarrow \alpha + \frac{n}{2}$$
$$\beta \leftarrow \beta + \frac{1}{2}\sum_{i=1}^{n}(x_i - \bar{x})^2 + \frac{n\kappa}{\kappa + n}\frac{(\bar{x} - \mu_0)^2}{2}$$

as described in [11]. Here, $\bar{x}$ is our observed mean difference in loss with individual observations $x_i$, and $n$ is the number of observations. Since the normal-gamma distribution follows:

$$\mathcal{N}\left(\mu, \frac{1}{\kappa\Gamma(\alpha, \beta)}\right)$$

we may use previously implemented Gaussian and Gamma sampling functions to achieve the same result as sampling directly from a normal-gamma distribution. This formulation indicates that six values must be tracked per model parameter: $\mu_0$, $\kappa$, $\alpha$, $\beta$, a running total of loss differences, and the sum of squares error (SSE) of these values and the sample mean. It would be possible to eliminate the last value if we instead retained every loss difference value, but simply tracking SSE is more efficient since we need only store one extra value, rather than one value per observation. The results of using Gaussian Sampling for MNIST and UCI Mushrooms can be seen below:

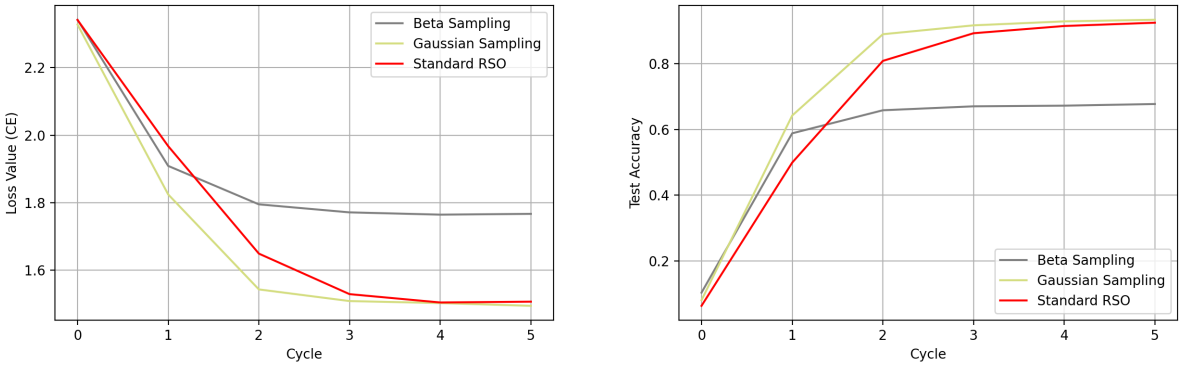


Figure 6: (Left) Loss plot for Gaussian Thompson Sampling, Beta Thompson Sampling, and standard RSO for the first 5 cycles of training. (Right) Test accuracy between the three versions in the same regime. It should be noted that Beta Sampling takes more than 5 cycles to reach the same level of loss and accuracy performance, but is also the fastest of the three in computation time.

So Gaussian Thompson Sampling is perhaps the most efficient gradient-free method we've looked at yet. It slightly edges out RSO on loss convergence and test accuracy and slightly under-performs against Binary Beta Thompson Sampling, but it is by far the fastest method. I believe that the slight lack of performance against the Beta sampling method as well as the speedy training time are both due to under-sampling of parameters. However, this under-sampling is not catastrophic and thus may ultimately be desirable with the addition of a slower, fine-tuning optimization is added to the end of training with it.

| **Approach** | Standard RSO | Beta Thompson Sampling | Gaussian Thompson Sampling |
|---|---|---|---|
| **Time per Cycle** | 23.6 min. | 13.4 min. | 7.1 min. |
| **Best Test Accuracy** | 92.4% | 84.8% | 93.3% |

## 4.3 Forgetting

One design choice for using Thompson Sampling for updates is how long to "remember" the results of selecting a particular parameter for update. In the previous sections I used the entire history of each parameter to inform the sampling distributions, but we could instead restrict the history to only the previous few events. If a specific parameter yielded bad results in the past, forgetting would allow such a parameter to build a better sampling distribution if it starts to yield good results later on.

One way to deal with this is to track the history in the *belief* as normal, but when the number of events exceeds the forgetting value, the oldest event is popped from the beginning of the list and the newest event gets added to the end. The only issue with this method of "forgetting" is that if a parameter builds a particularly bad distribution that causes it to rarely get sampled, it may never reach the point of forgetting, especially for longer memories. However, in this project I only deal with this more naive version and show that it is still sufficient. For brevity, I will sometimes refer to approaches which employ forgetting as "Name Fn," where "F" designates forgetting is used and "n" represents the length.

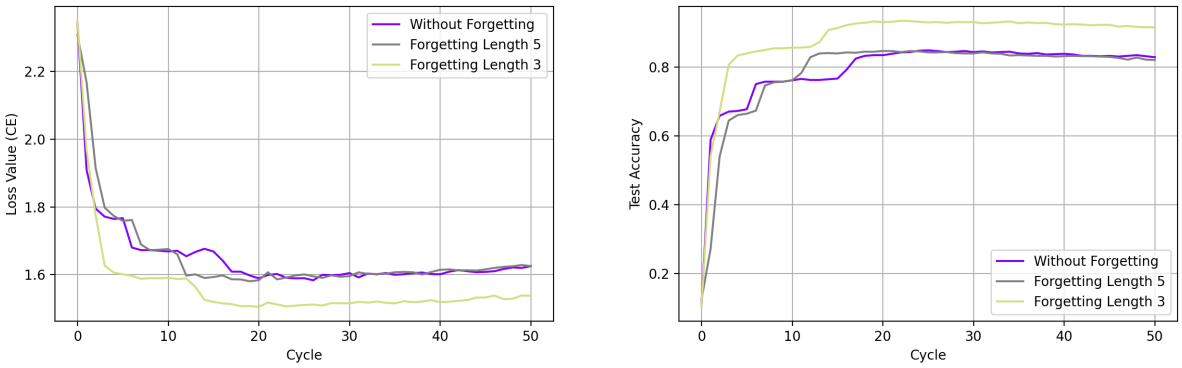


Figure 7: (Left) Loss plot for Beta sampling with forgetting of two different lengths, and without forgetting. (Right) Test accuracy between the three versions.

So there is a slight performance boost for length 5, but a substantial boost for length 3. This indicates that this simple version of forgetting work well in practice, though it remains a tuneable hyperparameter. This implementation gets significantly more complicated when Gaussian sampling is considered. The difficulty is that keeping a running memory for the purpose of forgetting is somewhat cumbersome and difficult due to the way the parameters aggregate (i.e. $\mu_0$ is not an arithmetic sum of the history). This would either require re-calculating the sampling parameters every time a piece of the history is forgotten —a computationally expensive workaround—or altering the *belief* structure significantly. In any case, I leave this as a direction for future work in order to constrain the scope of this project.

## 4.4 Binary Thompson Sampling

Now that we have seen the successes of using binary weights for RSO updates as well as the success of Thompson Sampling for parameter update selection, the natural next area to explore is that of the combination of the two.

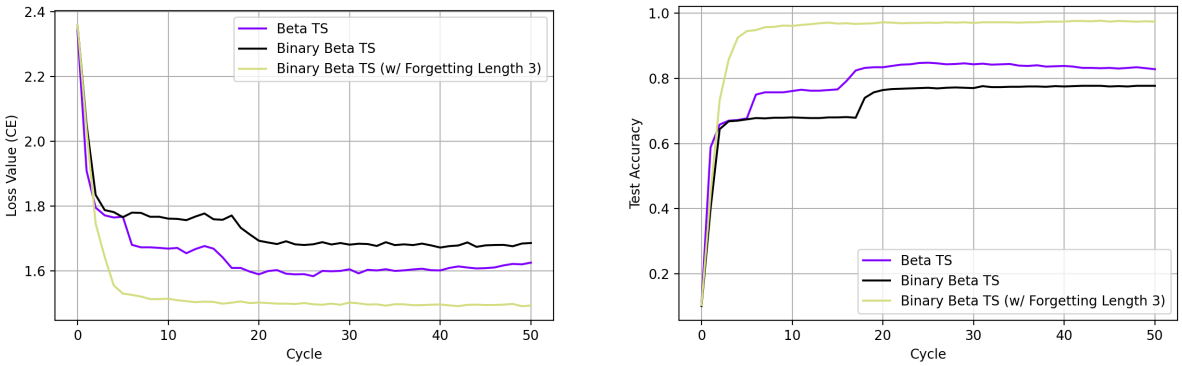


Figure 8: (Left) Loss plot for basic Beta sampling, binary Beta sampling, and binary Beta sampling with forgetting length 3. (Right) Test accuracy between the three versions.

| **Approach** | Standard RSO | Beta TS F3 | Binary Beta TS F3 |
|---|---|---|---|
| **Time per Cycle** | 23.6 min. | 13.4 min. | 2.35 min. |
| **Best Test Accuracy** | 92.4% | 93.4% | 97.7% |

Interestingly, there is a slight drop in performance when the approach is made binary, but would be satisfactory considering the significant train time decrease. However, when forgetting is introduced in the binary approach, the performance is actually increased over that of the basic Beta Thompson Sampling approach without an increase in training time, and the training itself is more stable than the other two. This makes binary Beta Thompson Sampling with forgetting the best performing gradient-free update method I explore in this project with every aspect considered. However, it may be possible to improve Gaussian sampling performance in the future, so I look at the results of the binary version of this as well:

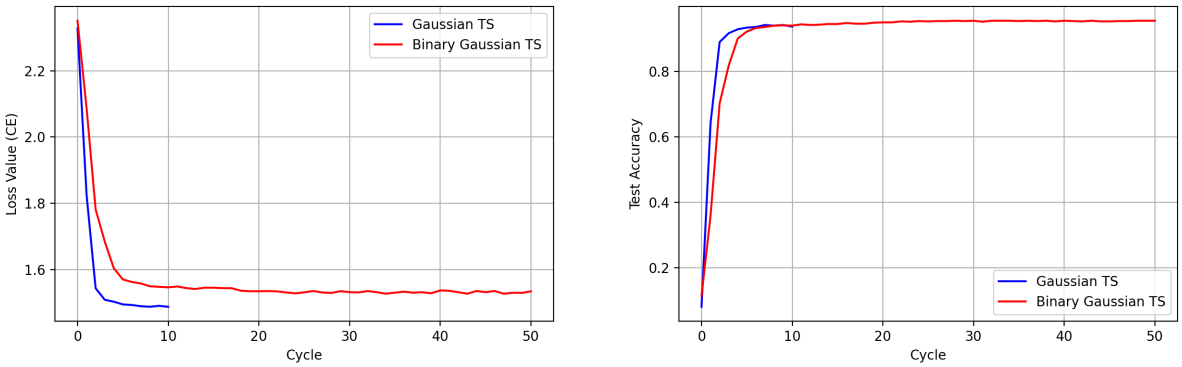


Figure 9: (Left) Loss plot for basic Gaussian sampling and binary Gaussian sampling. (Right) Test accuracy between the two versions. Due to training time constraints, only the first 10 cycles are shown for basic Gaussian sampling.

| **Approach** | Standard RSO | Binary Beta TS F3 | Binary Gaussian TS |
|---|---|---|---|
| **Time per Cycle** | 23.6 min. | 2.35 min. | 1.18 min. |
| **Best Test Accuracy** | 92.4% | 97.6% | 95.4% |

So Gaussian Thompson Sampling is perhaps the most efficient gradient-free method we've looked at yet. It slightly edges out RSO on loss convergence and test accuracy and slightly under-performs against Binary Beta Thompson Sampling, but it is by far the fastest method. I believe that the slight lack of performance against the Beta sampling method as well as the speedy training time are both due to under-sampling of

parameters. However, this under-sampling is not catastrophic and thus may ultimately be desirable with the addition of a slower, fine-tuning optimization is added to the end of training with it. As mentioned previously, with the current implementation it is not possible to use forgetting with the Gaussian version, however it may be interesting to see if this could even further improve performance in the future.

# 5 Conclusion and Future Work

In this project, I explored various new methods of optimizing deep neural networks without using gradient information. I showed various forms of Random Search Optimization and their efficacy, as well as an original implementation which is able to obtain good performance with a significant drop in training time from standard RSO. I also show that it is possible to further improve both training time and performance by using Thompson Sampling to select parameters to update by building sampling distributions over time based on previously observed loss results. Finally, I demonstrated the efficacy of various versions of this Thompson Sampling approach, including a more efficient binary-weighted approach and performance-improving method of only keeping a short-term memory of sampling statistics for each model parameter.

I saw quite a few surprising results during my work on this project, and some of these results were rather significant in the realm of gradient-free optimization. However, there are also plenty of avenues left to be explored. One area for improvement is to test these ideas with more difficult datasets, as I was largely constrained on time for my choices of data. Additionally, one of the major implementations not included in this project work is a method of forgetting for Gaussian Thompson Sampling, which will likely require a re-working of the *belief* structure in order to be efficient enough to be worthwhile. It also may be possible to make Gaussian Thompson Sampling in general a bit faster by using more efficient storage and sampling methods, as the current implementation is only a rudimentary proof-of-concept. And finally, I believe the ideas of Thompson Sampling I explored here are only the tip of the iceberg in the area of efficient parameter sampling, and that there may be better methods entirely that even determine contain better information about that the update values should be. I believe the approaches I demonstrated here are a good start, but that there is a lot more to be learned about how best to optimize a network for which you have no gradient information.

# 6 Appendix

## 6.1 Further Results: UCI MiniBooNE

The MiniBooNE dataset, hosted by the University of California - Irvine, consists of physical data from neutrino detector events. There are over 130,000 separate events with 50 real-valued features each. Every sample is labeled as one of two classes, either signal or background noise, and there are 36,499 signal events and 93,565 background events.

For this dataset, it is simpler to use a dense baseline model, rather than a convolutional model, due to the structure of the features being agnostic to adjacent features. This model is structured with four dense layers of width 100, 50, 50, and 2, all activated by ReLU except the last layer, which is activated by softmax. In order to keep the model similar to the convolutional baseline model used for MNIST, I also apply a 1-dimensional batch-norm layer after each internal dense layer. The results on this model via several approaches discussed in this project can be found below:

| **Approach** | **Time per Cycle** | **Best Test AUC** |
|---|---|---|
| Standard RSO | 3.78 min. | 0.857 |
| Binary RSO + BN | 2.27 min. | 0.831 |
| Beta TS | 2.28 min. | 0.858 |
| Binary Beta TS F5 | 28.7 sec. | 0.846 |
| Gaussian TS | 45.3 sec. | 0.865 |
| Binary Gaussian TS | 18.12 sec. | 0.852 |

For MiniBooNE, we don't see the same variability in test performance between each approach, likely due to the lower number of parameters, however we see similar differences in timing. Each approach performs roughly the same by test AUC, and all fairly well, but the latter three approaches had the fastest training times by a wide margin. This reinforces the conclusion that Thompson Sampling, especially in the binary case, can significantly reduce gradient-free training time while not suffering a drop in performance and, in some cases, even seeing performance improvement.

## 6.2 Further Results: UCI Mushrooms

The Mushrooms dataset, also hosted by UCI, consists of physical attributes of over 8,000 mushrooms, with 22 categorical attributes each. There are again just two classes for this dataset: edible and inedible, which consist 3,915 and 4,208 samples, respectively. The data are linearly separable, and thus a successful approach must reach an AUC of 1.0, and all of these models do. The more interesting component of the results are the timings, which can be found in the table below:

| **Approach** | **Time per Cycle** |
|---|---|
| Standard RSO | 6.37 min. |
| Binary RSO + BN | 3.27 min. |
| Beta TS | 2.29 min. |
| Binary Beta TS F3 | 38.2 sec. |
| Gaussian TS | 58.8 sec. |
| Binary Gaussian TS | 29.4 sec. |

It is clear from the results of all tests that the timings are significantly faster for Thompson Sampling, and especially for the binary Gaussian version. Since all approaches reached an AUC of 1.0, some of which did so in the first or second training cycle, this significant timing reduction is certainly without loss of performance here. Note that the baseline model for these results was also a dense model, identical to the MiniBooNe baseline model, except the first layer necessarily has fewer inputs to correspond to the fewer features of this dataset.

## 6.3 Aggregated Results for MNIST

In order to clearly see all of the results as they stacked up for MNIST, and to compare to the results for the other datasets, I've included all results for MNIST for the main approaches below:

| **Approach** | **Time per Cycle** | **Best Test Accuracy** |
|---|---|---|
| Standard RSO | 23.6 min. | 92.4% |
| Binary RSO | 9.43 min. | 45.9% |
| Binary RSO + BN | 9.4 min. | 97.6% |
| Ternary RSO | 20.0 min. | 43.4% |
| Beta TS | 13.4 min. | 84.8% |
| Beta TS F3 | 13.4 min. | 93.4% |
| Binary Beta TS F3 | 2.35 min. | 97.6% |
| Gaussian TS | 7.1 min. | 93.3% |
| Binary Gaussian TS | 1.18 min. | 95.4% |

# References


[1] L. Bottou and O. Bousquet, "The tradeoffs of large scale learning," in *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, (Red Hook, NY, USA), p. 161–168, Curran Associates Inc., 2007.

[2] R. Tripathi and B. Singh, "Rso: A gradient free sampling based approach for training deep neural networks," 2020.

[3] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[4] D. Dua and C. Graff, "UCI machine learning repository," 2017.

[5] D. Russo, B. V. Roy, A. Kazerouni, I. Osband, and Z. Wen, "A tutorial on thompson sampling," 2020.

[6] W. R. Thompson, "On the Likelihood That One Unknown Probability Exceeds Anther in View of the Evidence of Two Samples," *Biometrika*, vol. 25, pp. 285–294, 12 1933.

[7] W. Zhang, D. Zhou, L. Li, and Q. Gu, "Neural thompson sampling," 2020.

[8] X. Lu and B. V. Roy, "Ensemble sampling," 2017.

[9] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, "Weight uncertainty in neural networks," 2015.

[10] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," 2016.

[11] K. Murphy, "Conjugate bayesian analysis of the gaussian distribution," 11 2007.