# CS 6787 Final Report: Gradient-based Hyperparameter Optimization through Reversible Learning

**Shirley Kabir** [1] **Jing Rong Lim** [1] **Christopher Roman** [1]

## Abstract

Current machine learning literature propose several methods for hyperparameter search, which is a critical process significantly affecting a model's test performance. The most popular methods today are random search, grid search, Bayesian optimization, and sequential model-based optimization. However, despite how hyperparameters are manually chosen and tuned in a way similar to model parameters optimization, hyperparameter search remains vastly different from how model parameters are optimized. Hence, we build on Maclaurin's (Maclaurin et al., 2015) gradient-based approach for optimizing hyperparameters, where the loss function is differentiated with respect to the hyperparameters at each update step. We then compare performance and other system metrics with existing tuning methods. While Maclaurin developed autograd — an automatic differentiation library — to implement this algorithm, we opt to use PyTorch's automatic differentiation capabilities. For a logistic regression model trained on MNIST, gradient-based hyperparameter optimization had an decreased after 8-10 meta-iterations running on 2 epochs each. This approach also saw a 63MiB increase in memory usage, as well as more than 10x increase in average iteration time as compared to random and grid search. These results show that a gradient descent approach is viable for optimizing both the weights of a model and its hyperparameters, and should be further examined.

## 1. Introduction

In machine learning, gradient descent-based algorithms update model parameters by differentiating a loss function evaluated on training data with respect to the parameters- this differs from hyperparameter tuning, which modifies hyperparameters based on accuracy derived from validation data evaluated after training. Some examples of hyperparameters include learning rate, momentum constants, or even hidden layer size and number of layers in neural networks. Hyperparameter selection can impact the performance of a model significantly, potentially affecting the convergence rate of a model and final accuracy. As a result, hyperparameter tuning has evolved its own subfield in machine learning, with several different algorithms proposed and in use today.

### 1.1. Grid Search

Apart from manual selection, another common way to explore the hyperparameter search space is to simply generate all possible (viable) combinations of hyperparameters. 2 tunable hyperparameters with 2 possible choices each would generate 4 possible combinations of hyperparameters. All combinations of hyperparameters would then be used with the model, before storing the combination that produces the greatest validation accuracy.

### 1.2. Random Search

Random search was first proposed by Bergstra and Bengio (Bergstra & Bengio, 2012) as an alternative to the more common grid search and manual search algorithms at the time. Bergstra et al suggested that randomly sampling hyperparameters from the subspace could be a more efficient approach as compared to grid search. Since most hyperparameters do not have significant impact on model performance, random search allows a greater number of searches in the subspace of an important hyperparameter.

### 1.3. Gradient-based algorithms

However, both grid search and random search present a markedly different approach to hyperparameter tuning as opposed manual search. A typical manual search approach would have a machine learning practitioner try different combinations hyperparameters manually-if increasing a hyperparameter gave good results, a further search in that direction could yield better results. This manual search approach based on validation accuracy leads us to further

[1]Department of Computer Science, Cornell University, Ithaca, New York, USA.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

investigate gradient-based hyperparameter optimization as a viable approach.

Gradient-based optimization was first proposed by 1999 by Bengio (Bengio, 2000), and then further expanded by Domke (Domke, 2012) in 2012 as an approximate solve to bi-level optimization. This area of research explores the possibility of optimizing hyperparameters by estimating changes in validation loss with respect to hyperparameters. We choose to base the bulk of this paper on a further expansion of Domke's work by Maclaurin et al, titled "Gradient-Based Hyperparameter Optimization through Reversible Learning" (Maclaurin et al., 2015). As opposed to other gradient-based hyperparameter optimization using implicit differentiation (Pedregosa, 2016), Maclaurin's work expands on an iterative differentiation approach to Stochastic Gradient Descent, which can be applied towards non-convex functions.

**Implicit Differentiation** Pedregosa's (Pedregosa, 2016) approach towards gradient-based optimization utilized an 'implicit differentiation' algorithm. We can describe this gradient-based hyperparameter optimization problem as a doubly nested optimization problem; implicit differentiation refers to replacing the inner optimization problem with an implicit equation. Unfortunately, this requires some regularity conditions to be imposed, which makes this approach unapplicable for non-convex training.

**Iterative Differentiation** In contrast, Maclaurin's (Maclaurin et al., 2015) paper differentiates each step of the inner optimization algorithm (stochastic gradient descent), and uses the chain rule to aggregate and compute the gradients with respect to hyperparameters. Given the greater applicability of an iterative differentiation approach, we choose to compare Maclaurin's algorithm to other hyperparameter tuning methods.

## 2. Motivations

There are many reasons for us to replicate this work. The first reason is that, while the experimental results in (Maclaurin et al., 2015) is relatively robust, they do not try to make many direct comparisons between other hyperparameter optimization techniques like grid search and random search. We wanted to focus on making a direct comparison for particular ML problems. The second reason is that the algorithm is written using autograd, which is a somewhat outdated automatic differentiation library. By implementing the algorithm in PyTorch without extreme difficulty, we open up new opportunities to ML practitioners in research and industry who are likely very familiar with the PyTorch.

Below, we list the two main algorithms we implement which are originally presented in (Maclaurin et al., 2015).

**Algorithm 1** Stochastic gradient descent with momentum

**input:** initial $\boldsymbol{w}_1$, decays $\boldsymbol{\gamma}$, learning rates $\boldsymbol{\alpha}$, loss function $L(\boldsymbol{w}, \boldsymbol{\theta}, t)$
initialize $\boldsymbol{v}_1 = \mathbf{0}$
**for** $t = 1$ **to** $T$ **do**
  $\boldsymbol{g}_t = \nabla_{\boldsymbol{w}} L(\boldsymbol{w}_t, \boldsymbol{\theta}, t)$
  $\boldsymbol{v}_{t+1} = \gamma_t \boldsymbol{v}_t - (1 - \gamma_t)\boldsymbol{g}_t$
  $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha_t \boldsymbol{v}_t$
**end for**
**output** trained parameters $\boldsymbol{w}_T$

**Algorithm 2** Reverse-mode differentiation of SGD

1: **input:** $\boldsymbol{w}_T$, $\boldsymbol{v}_T$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$, train loss $L(\boldsymbol{w}, \boldsymbol{\theta}, t)$, loss $f(\boldsymbol{w})$
2: initialize d$\boldsymbol{v} = \mathbf{0}$, $d\boldsymbol{\theta} = \mathbf{0}$, $d\alpha = \mathbf{0}$, $d\gamma = \mathbf{0}$
3: initialize d$\boldsymbol{w} = \nabla_{\boldsymbol{w}} f(\boldsymbol{w}_T)$
4: **for** $t = T$ **counting down to** 1 **do**
5:  $d\alpha_t = d\boldsymbol{w}^\top \boldsymbol{v}_t$
6:  $\boldsymbol{w}_{t-1} = \boldsymbol{w}_t - \alpha_t \boldsymbol{v}_t$
7:  $\boldsymbol{g}_t = \nabla_{\boldsymbol{w}} L(\boldsymbol{w}_t, \boldsymbol{\theta}, t)$
8:  $\boldsymbol{v}_{t-1} = [\boldsymbol{v}_t(1 - \gamma_t)\boldsymbol{g}_t]/\gamma_t$
9:  $d\boldsymbol{v} = d\boldsymbol{v} + \alpha_t d\boldsymbol{w}$
10:  $d\gamma_t = d\boldsymbol{v}^\top(\boldsymbol{v}_t + \boldsymbol{g}_t)$
11:  $d\boldsymbol{w} = d\boldsymbol{w} - (1 - \gamma_t) d\boldsymbol{v} \nabla_{\boldsymbol{w}} \nabla_{\boldsymbol{w}} L(\boldsymbol{w}_t, \boldsymbol{\theta}, t)$
12:  $d\boldsymbol{\theta} = d\boldsymbol{\theta} - (1 - \gamma_t) d\boldsymbol{v} \nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{w}} L(\boldsymbol{w}_t, \boldsymbol{\theta}, t)$
13:  $d\boldsymbol{v} = \gamma_t d\boldsymbol{v}$
14: **end for**
15: **output** gradient of $f(\boldsymbol{w}_T)$ w.r.t $\boldsymbol{w}_1$, $\boldsymbol{v}_1$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$, $\boldsymbol{\theta}$

The algorithms presented have some ambiguity, so we make some assumptions about how to resolve it. Firstly, in Algorithm 1 it is unclear exactly what $T$ is. Depending on the interpretation, it could be the number of epochs (i.e. the number of passes over the **entire** dataset), or the number of minibatches **times** the number of minibatches in the dataset. Because Algorithm 1 is simply describing SGD with momentum wherein gradients are computed per minibatch, we decide to have $T$ be the latter. In Algorithm 2, it is unclear what exactly the shapes of the variables are supposed to be. We need to assume that $\boldsymbol{w}_t$, $\boldsymbol{v}_t$, $d\boldsymbol{v}$, $d\boldsymbol{\theta}$, and $d\boldsymbol{w}$ are $n-$dimensional **vectors**. Additionally, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$, $d\boldsymbol{\gamma}$, and $d\boldsymbol{\alpha}$ are vectors of length $t$, and a subscript of $t$ on these variables means to index into the $t$-th element of the vector.

Confusingly, the experiments done in (Maclaurin et al., 2015) has $\boldsymbol{w}$ as a rank-5 tensor instead of simply a vector (i.e., rank-1 tensor). This is strange because it doesn't make much sense to transpose a tensor and multiply it by another, like in line 5 of Algorithm 2. To solve this, we assume that the authors must have flattened the tensors like $\boldsymbol{w}$ everywhere in Algorithm 2, except for when we compute the loss $L(\boldsymbol{w}_t, \boldsymbol{\theta}, t)$ where $\boldsymbol{w}_t$ must be treated as a tensor for the loss and RMD to make sense.

## 3. Implementation

We decided to use Python 3 and PyTorch v1.3.1 to develop Algorithm 1 and 2. We chose PyTorch because in comparison to TensorFlow v2 it had better documentation. We also wanted to ensure that we could take the hessian vector product and there were a good number of examples of code that could perform this operation.

### 3.1. PyTorch

The original paper's code referenced a library called `autograd` which has methods that can automatically differentiate native Python and Numpy code in addition to performing useful operations like reverse-mode differentiation which is the focus of this paper. Since autograd was also implemented by those who implemented the code for this paper we wanted to perform the same operations on tensors using PyTorch methods.

For Algorithm 1, we implemented `compute_gradient()` which is used in `stochastic_gradient_descent()` (Algorithm 1) which takes in the weight tensor, features, labels, and the loss function. In order to take the gradient of the loss function with respect to the weights, we needed to do $weights * features$ where weights is a tensor which has the flag `requires_grad=True` then run the loss function on that and the labels. To generate the backward graph, `loss.backward()` which essentially populates the gradients, but only for the nodes which have both `requires_grad` and `is_leaf` True. Gradients are of the output node from which `.backward()` is called, w.r.t other leaf nodes (Kumar, 2019). `weight.grad` then holds the value of gradient which in our case will hold $\partial loss/\partial w$.

For Algorithm 2 we had to implement the Hessian vector product $\nabla_{\boldsymbol{w}} \nabla_{\boldsymbol{w}} L(\boldsymbol{w}_t, \boldsymbol{\theta}, t)$. Which consisted of taking the dot product of the gradient with the vector $d\boldsymbol{v}$ and then taking the partial derivative with respect to w again. This was done in `hessian_vector()` where we took the dot product using `torch.dot` followed by `.backward(retain_graph=True)` which took the backward of this dot product again, but with `retain_graph` set to True which means that the graph used to compute the grad won't be freed and can be reused for further computations, then ran `weight.grad` again to get the Hessian vector product.

### 3.2. Details

For **grid search**, we generated two tensors that represented a list of gamma and alpha values. We then passed those values onto `stochastic_gradient_descent()` which then returns a weight vector and that is then passed into `accuracy()` which returns the accuracy rate. Lastly, we looped through the gamma and alpha values and kept track of what had the best accuracy and only updated the hyper parameters if it beat the running `best_accuracy`.

For **random search**, we wrote a method called `generate_random_hyperparameters()` which generates hyperparameters within a given range (we selected [0.01, 0.03] for gamma and alpha). Given those tensors, we run the same thing as grid search to retain the 'best' hyper parameters.

The paper includes Algorithm 3 which was meant to correct the issue of finite numerical precision. This occurs as a result of all of the operations required to reverse the operations in forward SGD. Line 8 in Algorithm 2 `v(t-1) = [v(t) + (1-gamma(t))g(t)]/gamma(t)` raises the issue of losing information due to the division and multiplication of a relatively small decimal. We didn't have to implement algorithm 3 because we already run SGD on the same values so we retain the values at each step and generate a value called `all_parameters` that contains the weight, velocity, and gradient tensors (i.e. $g_t$, $w_{t+1}$, and $v_{t+1}$ from Algorithm 1).

### 3.3. Difficulties

When implementing Algorithm 2, we encountered a road bump where when we did the matrix dot product of $\partial loss/\partial w$ and $d\boldsymbol{v}$, we ended up with a 10x10 matrix, since 10 is the number of classes, instead of a scalar which is necessary in order to run `.backward()` on that tensor. We ended up flattening both the vector and did the dot product of that in order to get a scalar and only flattened the values necessary to perform this dot product effectively.

More importantly, we ran into several issues with running `autograd.grad` and `autograd.backward`. Although there were several examples of these methods being used, most of them didn't use it on the tensor directly. We ran into issues such as not being able to take the gradient with respect to the weights because it was updated somewhere else in the process since tensors keep track of all of the operations that modifies it like cloning or adding. Whenever we're running `x.grad` x must always be a tensor with `requires_grad` set to True. When we cloned it, it no longer was just that. So we had to run `with torch.no_grad()`. In this mode, the result of every computation will have `requires_grad=False`, even when the inputs have `requires_grad=True` (tor).

## 4. Experimental Results

We next run a number of experiments to determine the performance of gradient-based hyperparameter optimization as compared to other common hyperparameter tuning methods.

*Table 1.* Best Accuracy, Training Loss, and Test Loss Across Hyperparameters

| | GRADIENT-BASED | **RANDOM** | GRID |
|---|---|---|---|
| ACCURACY | 91.48% | **91.85%** | 89.60% |
| TRAINING LOSS | 0.2656 | **0.2325** | 0.3334 |
| TEST LOSS | 0.2348 | **0.2235** | 0.3173 |

We run our implementation of each algorithm in Python on the MNIST dataset, where the inner optimization function performs stochastic gradient descent for logistic regression. We then record several metrics such as accuracy, CPU time, and memory usage for all three algorithms.

### 4.1. Accuracy

**Hypothesis**: Using a gradient-based approach to optimize the learning rate schedule will result in a higher accuracy than using random search or grid search for image classification.

**Important Metrics**: We will keep track of the training loss, test loss, and accuracy for each meta-optimization step.

**Protocol**: We will replicate one of the experiments done in the paper which trains a logistic regression model on MNIST and hyper-optimizes the learning rate schedule. We do this because the paper doesn't do a direct comparison against random search or grid search.

**Results**: We can see in Figure 1 and 2 how the loss and accuracy changes per meta iteration. Each meta iteration represents a change to the hyperparameters after running Algorithm 1 and 2. Because we are using a gradient based approach, we expected that the loss and accuracy would decrease as time progressed. However, it seems that there is no improvement to the accuracy or loss, and even potentially gets worse over time. It is possible that our implementation and assumptions that we made are incorrect, which leads us to be unable to reproduce some results in Maclaurin's paper.

In Table 1 we compare the accuracy, training loss, and test loss across gradient-based optimization, random search, and grid search. In this experiment, random search does the best, with gradient-based optimization being the next best, and grid search being the worst. It is important to note that with the gradient-based optimization, we set the hyperparameters to be randomly initialized, which started the accuracy in the beginning relatively high. So while some improvement was seen in accuracy in Figure 2, it is not very significant.

### 4.2. Time

**Hypothesis**: In order to store and compute the hypergradient, using this gradient-based approach will have an increase in CPU clock-time clock-time as compared to random search and grid search.


*Figure 1.* The training and test loss per meta iteration

**Proxy statement**: CPU time per set number of iterations

**Protocol**: We calculate an average CPU time per meta-iteration by using Python's in-built time module to calculate CPU time.

**Results**: For all experiments, we observe that gradient-based hyperparameter tuning requires more than 10x the time random search and grid search requires.

### 4.3. Memory

**Hypothesis**: In order to store and compute the hypergradient, using this gradient-based approach will have an increase in memory usage and per iteration as compared to random search and grid search.

**Proxy statement**: Memory usage for each algorithm

**Protocol**: On each experiment, we examine memory usage for gradient-based optimization, random search, and grid search. We complete this by using the Python library memory-profiler to monitor memory consumption of each function.

**Results**: By analyzing memory performance for a single meta-iteration of the random search, grid search, and gradient-based algorithms, we observe that gradient-based

*Table 2.* Accuracy Values Between Noisy, Clean, and Original MNIST Dataset

| TRAIN DATA | TEST DATA ACCURACY | $Epoch_1$ ACCURACY | FINAL VALIDATION ACCURACY | TRAINING ACCURACY |
|---|---|---|---|---|
| (BASELINE) NOISY | NOISY | 82.97% | 85.74% | 88.28% |
| CLEAN | NOISY | 86.63% | 87.61% | 86.20% |
| CLEAN | CLEAN | 93.69% | 96.47% | NAN |
| ORIGINAL | ORIGINAL | 90.08% | 90.92% | NAN |
| SPLIT | SPLIT | 90.49% | 91.14% | NAN |

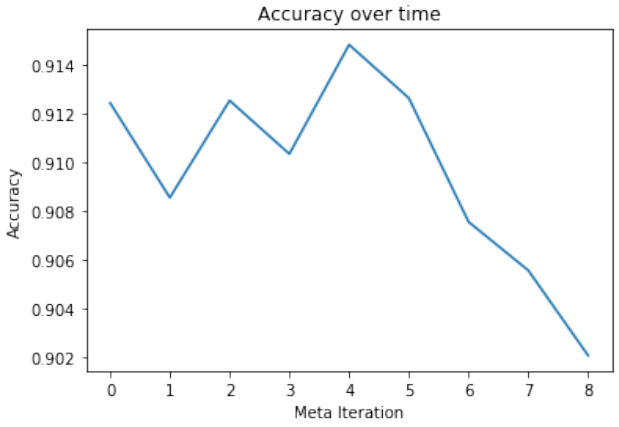


*Figure 2.* The accuracy per meta iteration

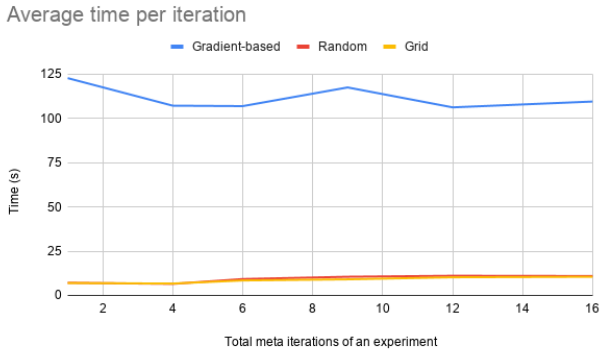


*Figure 3.* Average time per meta iteration, for experiments run for x meta iterations. Each gradient-based, random, and grid search algorithms were run with 2 epochs and batch size of 64

hyperparameter tuning required about 109MiB extra memory for the tuning process, as opposed to a negligible amount for grid search and random search. However, we also note a drop in the measured SGD memory for gradient-based tuning as opposed to grid search and random search; we hypothesize that this is due to the variables returned by SGD reused in the gradient-based tuning algorithm, which would normally be calculated instead of stored in the original algorithm. As a result, we readjust the increase in memory to be approximately 63MiB.

### 4.4. Noisy Data

**Hypothesis**: Splitting training values for each hyperparameter instead of having all hyperparameters learn from the same training set will reduce the effect of noisy data on the accuracy of the model.

**Proxy statement**: Observing the training versus validation errors from the model.

**Protocol**: Given a noisy data set, we are going to split into training and validation data set in an unbiased fashion (i.e. leaving them to be equally noisy). We will then train our algorithm and store the training and validation error rates for a baseline comparison. We will then have a clean version of the data set by removing certain values to make noise almost nonexistent. Then train the model on this data set and validate against noisy data to check which examples have a wrong label assigned to them, and store the error rates for this procedure. Lastly, we are going to assign one hyperparameter to a set of training samples in order to reduce the overall effect of noise and validate against the clean data and return those error values in order to make a comparison against the other 2 sets of error values.

**Expected Results**: The training error for the first run (baseline) should be approximately equal to the validation error, given that they are both dealing with noisy data. The second run (oracle) should have a much lower training error than validation error since the validation error is meant to be higher to track which data points are affected by noise. The last run (data hyper-cleaner) should have a lower validation error than training error given that it is more generalized than the training values. $E_{TR(BASELINE)} \geq E_{TR(CLEAN)}$ and $E_{VA(BASELINE)} \geq E_{VA(CLEAN)}$.

**Results**: When the original training data was split so that $\gamma$, and $\alpha$ were learned form different data sets there was an improvement in accuracy given the same initial alpha, gamma, and weights by approximately 0.22%. Additionally,


*Figure 4.* Memory usage per algorithm. Each gradient-based, random, and grid search algorithms were run with 1 epochs and batch size of 64

the expected results for the error rate comparison between baseline (noisy train and validation) vs. clean (clean train and noisy validation) were supported by the experiments. (Refer to Table 2)

## 5. Conclusion

Despite our initial hypothesis that gradient-based hyperparameter optimization could provide a quicker approach to finding optimal hyperparameters, it appears that the huge system performance hits should require a significant accuracy improvement to justify the use of the algorithm. Unfortunately, even without considering system performance, the measured accuracy after 8-10 meta iterations showed that gradient-based hyperparameter performs worse than random search or grid search.

These experiments ultimately show that this gradient-based approach is quite difficult to implement and may not provide very much improvement over standard methods like random search and grid search. In Maclaurin's paper, they question the meaning of having so many hyperparameters, and whether or not the model overfits. However, the experiments do show that PyTorch is a powerful enough framework to implement various algorithms presented in ML research.

We tested against clean and noisy data sets in order to prevent the algorithm from being too optimistic. We additionally split the dataset and trained each hyperparameter on a different parts in order to optimize the accuracy rate and prevent over-fitting.

## References

Automatic differentiation package - torch.autograd. URL https://pytorch.org/docs/stable/autograd.html.

Bengio, Y. Gradient-based optimization of hyperparameters. *Neural Comput.*, 12(8):1889–1900, August 2000. ISSN 0899-7667. doi: 10.1162/089976600300015187. URL http://dx.doi.org/10.1162/089976600300015187.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization, 2012.

Domke, J. Generic methods for optimization-based modeling, 2012.

Kumar, V. Pytorch autograd, 2019. URL https://towardsdatascience.com/pytorch-autograd-understanding-the-heart-of-pytor

Maclaurin, D., Duvenaud, D., and Adams, R. P. Gradient-based hyperparameter optimization through reversible learning, 2015.

Pedregosa, F. Hyperparameter optimization with approximate gradient, 2016.