Quantifying Uncertainty of DNN Hyperparameter Optimization using a First Order Type

Sayem Mohammad Imtiaz, Sumon Biswas and Yuepei Li

Iowa State University, Ames, IA, 50011, USA
sumon@iastate.edu, sayem@iastate.edu and liyp0095@iastate.edu

Abstract. Hyperparameter optimization is a difficult problem in developing deep learning applications. Recently, random search based strategies have been proven efficient for optimizing hyperparameters. However, programmers can not overtly represent uncertainty of the chosen hyperparameter values and accuracy of the model while performing a random search. In this project, we utilize a first order type Uncertain<T> to approximate the distributions of the hyperparameters so that programmers can pick values with certain confidence. This type helps us to represent uncertainty of random hyperparameters and allows us to easily propagate this uncertainty through computations, perform statistical tests on distributions without resorting to complicated statistical concepts, and determine uncertain hyperparameter value in required significance level.
To the best of our knowledge, there has not been any attempt to introduce the probabilistic programming concept in DNN hyperparameter optimization. The contributions of this project are as follows. First, we have implemented the first order type Uncertain<T> to hold the distribution of loss values over the randomly chosen hyperparameters. The main goal is to help programmers overtly represent uncertainty in chosen hyperparameters and make conditional statements using that. By using this type, we define algebra over random variables so that the uncertainty of the hyperparameters can propagate through the calculations and provide convergence speed and increase in accuracy. Second, our method performs significantly better than the random search method which is used by most of the DNN libraries. Our result shows that while 62% of the random search trials fall below the accuracy threshold, only 23% time our method fall below the threshold.
Keywords - Uncertainty, DNN, hyperparameter, optimization, machine-learning

1 Introduction

Uncertainty plays a fundamental role in machine learning (ML) algorithms and applications. Given the data, often we estimate the model parameters, predictions on the data and future consequences. The difference between the estimated value and the original value is referred as uncertainty. For example, if we want to model the bias of a coin by observing independent toss outcomes, only a few observations will not provide good estimate of the original distribution. Probability theory provides a framework to quantify uncertainty in machine learning models and predictions [1][2].

However, uncertainty in machine learning models is not overtly represented by the programmers. For instance, while using deep neural networks (DNN) for solving pattern recognition problems such as image classification or speech recognition, programmers use discrete types (e.g., integer, double, boolean) to handle uncertain data. At

a lower level uncertainty can originate from the measurement noise of the data such as blur pixels in images. At a higher level, model structure (e.g., neural network or linear regression) and model parameters (e.g., number of neurons in DNN) can be uncertain. Programmers might work with uncertainty in adhoc ways but since ML tasks are complex and data-intensive, often the inherent uncertainty in the model parameters is ignored. In this context, probabilistic programming can useful to overtly represent uncertain data and perform operations on them.

Recent development in probabilistic programming languages let the programmers express probabilistic model as programs [3][4]. The programs can manipulate data without losing latent probability distributions. The probabilistic models written using probabilistic programming languages show great promise to explicitly represent uncertainty in ML algorithms [5]. Programmers can develop an inference engine to simulate the model and generate values of the unobserved variables based on the observed data. By performing Monte-Carlo sampling strategy on the simulated variables, we can quantify uncertainty of the model.

One of the most difficult problem in building neural network model is that there is no hard and fast rule for setting the hyperparameter values of the model. Given a dataset and a problem, if we want to build a neural network to solve the problem, we have to consider a number of uncertain hyperparameters. For instance, network architecture (number of layers and hidden units), loss function, learning rate etc. can be uncertain. Two commonly used methods for optimizing hyperparameters are manual search and grid search [6][7]. Popular machine learning libraries (e.g., scikit-learn[1]) also use similar strategy for hyper parameter optimization. However, recent studies show that random search is more efficient in terms of processing time than grid search when number of hyperparameters increases [8]. Since grid search adapts an exhaustive strategy on all possible values of a hyperparameter which is not very useful in models having many hyperparameter such as in deep learning.

Hyperparameter optimization performance also varies depending on the domain and dataset. Recently, Stefan et al. proposed and robust and scalable hyperparameter optimization method [9] compares their performance with three other state-of-the-art techniques, which are Bayesian optimization, Hyperband and Random Search. The main goal of our method is to allow DNN programmers to control the hyperparameter search over the accuracy. We have represented the uncertainty of the model performance with randomly chosen values. Then we have also made performance comparison of our method with random search.

In this project, our goal is to aid the deep learning programmers to quantify uncertainty in the model hyperparameters and make an informed decision while initializing hyperparameters. A problem with the random search is that it doesn't take uncertainty of random hyperparameters into account while picking a best value which may adversely impact models when trained on the different distribution of the input domain. To that end, we have leveraged a first order type Uncertain<T> to represent the uncertainty in the random hyperparameters and choose best value by performing statistical tests on the distribution. The main contributions of the project are:

[1] http://scikit-learn.sourceforge.net

1. We have introduced the probabilistic programming concept in DNN hyperparameter optimization.
2. We have utilized a first order type Uncertain<T> to approximate the distributions over the possible hyperparameter values.
3. Describe the algebra to perform computations over the uncertain hyperparameters.
4. Provide syntax to ask boolean question on the uncertain data type to control false positive and false negative.
5. Improve the performance of random search for hyperparameter optimization.

The rest of the report is organized as follows. In section 2, we have described the background and related works. In section 3, we have formulated the problem statement, motivated the problem using an example and illustrated the solution sketch. Finally, in section 4, we have described our future plan.

2 Related Work

The topic of uncertainty in machine learning is raised for a long time [10]. However, with the recent success of artificial intelligence, recently more research techniques have been presented.

For example, Zhang et al. [11] studies the uncertainty in *k*NN classification. They introduced a certainty factor measure based strategy and demonstrate that this strategy outperforms standard *k*NN classification in accuracy. Some other researchers [12][13][14][15] studies uncertainty in neural network (NN). In particular, Blundell et al. [14] introduced an algorithm, called *Bayes by Backprop*, for learning the distribution of uncertainty on each back propagation. They demonstrated that weighted uncertainty can improve generalisation in non-linear regression problems and help to make the trade-off between exploration and exploitation in reinforcement learning. Gal et al. [12] developed a framework about dropout training in deep NN in 2016, which approximates Bayesian inference. By estimating the dropout's certainty, Gal et al. reduced the complexity caused by Bayesian model and improved the predictive performance. Gal et al. [13] further leveraged the Bayesian deep learning in active learning framework in 2017 and obtained a significant improvement on existing active learning approaches. Lakshminarayanan et al. [15] proposed a simple, parallelizable and scalable method which requires very little hyperparameter tuning and outperforms approximate Bayesian NNs in high quality predictive uncertainty estimates.

Uncertainty also been studied in Programming Language domain. In 2014, Bornholt et al. [5] proposed a new programming abstraction called Uncertain<T> to tackle the uncertainty in commonly used programming languages, such as *C++*, *C#*, and *Java*. This abstraction reduced the developer's work by letting them use the abstraction directly, instead of writing codes of statistics by themselves. In 2015, Bornholt et al. [16] further improved the programmability and accuracy. They provided new program constructs to specify application specific domain knowledge, which they named context, and implemented a runtime which automatically composes context. They demonstrate their constructs are necessary for Internet of Thing (IoT) programming and easy to use for developers. Boston et al. [17] discussed uncertainty in approximate computing.

They provided a type system to captures the error probability and bounds deviation of each expression from the correct value. They demonstrated their system is convenient and low annotation burden. They also discussed the problems in approximate hardware design. Nandi et al. [18] discussed new types of errors in probabilistic programs and introduced a programming model (FLEXI), based on Decorated Bayesian Network, and a tool (DePP) to help developers to detect errors.

Some researchers try to combine deep learning and probabilistic programming languages together. They name the new concept deep probabilistic programming languages. Ghahraman et al. [19] introduced the probabilistic framework and reviews the advances in probabilistic programming. They emphasize the importance of representing and manipulating uncertainty about models and predictions in many fields, including deep learning. Another study [3] explained the concept of deep probabilistic programming languages and characterized their strengths and weaknesses.

In big data, Agarwal et al. [20] investigated how to build a fast and reliable approximate query processing system. They implemented a query approximation pipeline to obtain "close-enough" answers at interactive speeds. Manousakis et al. [21] presented a framework for uncertainty propagation (UP) on a directed acyclic graph(DAG). They allow developers to develop MapReduce codes simulating specific DAG that uncertainties propagate in it. By this high-level abstraction, they handled data with uncertainty with high accuracy.

Esfahani et al. [22] discussed the uncertainty of self-adaptation software systems. Self-adaption is the ability that makes different actions or executions according to different contexts. They introduced a novel approach, based on possibility theory, called POssIbilistic SElf-aDaptation (POISED), to tackle the uncertainty problem in self-adaption decisions. POISED calculates both the positive and negative outcomes of uncertainty and choose the execution that maximum the best range of potential behavior. Perez et al. [23] introduced the concept of uncertainty in self-adaption system. They proposed a definition of uncertainty in computer science, together with the a taxonomy of different uncertainty models. They also analyzed possible sources and existing reduction methods of uncertainty in self adaptive software.

3 Approach

In this section, we formally introduce the problem and describe it using a motivating example. Then, we outline the solution approach building upon Uncertain<T> framework [5]. Finally, we describe the future plan to implement the approach.

3.1 Problem Statement

Optimizing hyperparameters (e.g., learning rate, number of hidden units etc.) is very crucial in ML algorithms for minimizing loss function and generalization errors, and fast convergence. To that end, an efficient technique has been proposed, known as random search, to optimize hyperparameters [8].

Random search draws random samples from the distribution of a hyperparameter uniformly and determines the sample which produce best result for the given loss func-

tion. Formally, if λ is a hyperparameter, then random search draws uniformly from possible trials of $\lambda = \{\lambda_1, \lambda_2, \lambda_3, ...\}$ and minimizes loss function:

$$\lambda^* = \underset{\lambda}{\operatorname{argmin}}\, Loss_\lambda(x) \tag{1}$$

As we can see from Equation 1, random search picks the trial of λ which minimizes the loss function most. However, that particular choice of hyper-parameter might not work optimally on distribution of all inputs, X. In addition to that, combination of hyperparameters and variables in the algorithm, for instance, weights in neural network, may lead to different result for this choice of trial. Therefore, we need to quantify the uncertainty, which may be introduced due to different combination of other parameters, in picking best trial from random search. Instead of asking whether this particular choice of trial is the best, we can ask, how much evidence is there for this trial being the best.

3.2 Motivating Example

Gradient descent algorithm is widely used as an optimizer in deep neural networks. Using gradient descent, we try to find the optimal set of weights, biases, or other hyperparameter values for which the loss is minimum. Suppose, we are trying to optimize the hyperparameters in Table 1. A gradient descent works by moving downhill of the loss function curve. However, a loss function curve for a certain problem might not have only one global minimum point. It might have other local minimum points as well as can be seen in Fig. 1. D is a local minimum in the curve shown in Fig. 1 where the slope of the curve is 0. Therefore, if weights or other dependent parameters are initialized somewhere in left of the local minimum, D, the gradient descent will stop moving once it reaches point D as slope becomes 0. However, if weights are initialized in somewhere in immediate left downhill of the global minimum, the algorithm will be able to reach the global minimum point.

Hyperparameter	Description	Value Range
n_hidden_layers	number of trees in forest	[1, 10]
n_hidden_units	max number of features considered	[1, 10]
learning_rate	the rate that NN learns from errors	[0.01, 0.50]
activate function	the function used to pass weighted sum	[`sigmoid`, `relu`, `tanh`]
dropout	the ratio that NN drop neurons in each layer	[0.0, 0.5]

Table 1: Hyperparameters used in neural network and their ranges

As a result, uncertainty in random initialization of the dependent parameters may have varying impact on the cost function. Therefore, quantifying this uncertainty to make an informed initialization of the dependent parameters is very important.

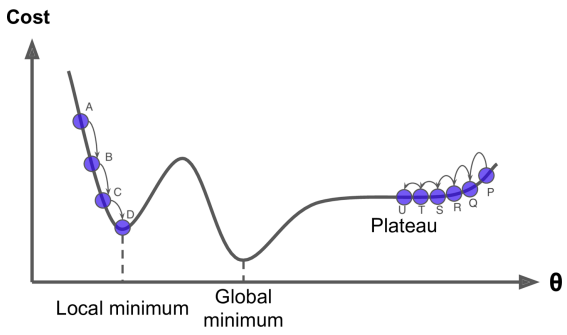


Fig. 1: Optimizing the value of hyperparameter θ for which the cost is minimum [24]

3.3 Solution Approach

In this section, we will leverage a first order type Uncertain<T> [5] on random search to quantify uncertainty in hyperparameter optimization and select the best choice. As an illustration, we work on optimizing a hyperparameter i.e., number of hidden units in the network.

Uncertain<T> is a first-order data type which encapsulates a distribution of numeric type T such as `Integer`, `Double` etc. It adopts a lazy technique to evaluate the conditionals to generate concrete values over the distribution. It does so by approximating a sample distribution over the real distribution while acting on conditionals or concrete values. As a result, it introduces an accuracy vs efficiency trade-off [5].

Number of hidden units in the network is an uncertain hyperparameter. The variance of number of hidden units affects the outcome of a network in many ways. Moreover, a network working optimally for a particular number of hidden units does not necessarily imply it is optimal for all other networks. Therefore, choosing a optimal number of hidden units in neural network is crucial for producing the best result.

```
1 Uncertain<Integer> drawHiddenUnitSample(Integer
    minHiddenUnit, Integer maxHiddenUnit) {
2     Func<Integer> SamplingFunction = () => {
3         //randomly draw one sample from uniform
    distribution
4         return RandomUniform(minHiddenUnit, maxHiddenUnit
    );
5     }

7     //pass reference to sampling function
8     return new Uncertain<Integer>(SamplingFunction);
9 }
```

Listing 1.1: Sampling the uniform distribution of hidden unit

```
Uncertain<Float> drawLossFunctionSample(Integer
    numberHiddenUnit) {
    Func<Float> SamplingFunction = () => {
        //train a user-supplied NN model
        //with specified number of hidden units
        //and obtain value of the loss function
        NNModel=randomIntialization();
        Float lossValue=NNModel(numberHiddenUnit);

        return lossValue;
    }
    //pass reference to sampling function
    return new Uncertain<Float>(SamplingFunction);
}
```

Listing 1.2: Sampling loss function distribution

```
Uncertain<Integer> hiddenUnitDistribution =
    drawHiddenUnitSample(1, 100);

AcceptableLossValueThreshold= 0.3;
Integer iteration = 100;

for(Integer counter = 0; counter <= iteration; counter++)
{
    //draw a sample from hidden units distribution
    Integer hiddenSample = hiddenUnitDistribution();

    //get a ref.  to loss function distribution for each
    hidden sample
    Uncertain<Float> LossDistribution =
    drawLossFunctionSample(hiddenSample);

    //perform a hypothetical test on loss function
    //distribution to determine there are sufficient
    evidence
    //for loss within the acceptable range
    if( LossDistribution<=AcceptableLossValueThreshold)
        //report a acceptable hidden unit size found.
}
```

Listing 1.3: User program

Random search typically takes a list of possible hyperparameter values and trains the model based on a set of randomly chosen values from the given list. For instance, for the problem we are considering, the developer needs to supply a list of possible hidden

unit sizes to the random search. Then it returns most viable value, chosen randomly from the list, which produces the possible best result for the model.

In our approach, we instead draw a hidden unit size from a uniform distribution of possible hidden unit sizes, which is encapsulated with a Uncertain<T> data type. Listing 1.1 is a pseudo-implementation for drawing random sample for hidden unit size. In line 1 of the listing 1.3, a reference to the sampling function of hidden layer size is obtained. As we can see, in line 12 of the same listing, the obtained reference to the sampling function is being called as many times as needed which returns a sample hidden unit size upon each invocation. In this example, although we are drawing 100 random samples of hidden unit, a developer can draw as many as they need.

Next, in line 13 of listing 1.3, a reference to the sampling function for loss function distribution is obtained. Listing 1.2 provides a pseudo-implementation for approximating loss function distribution. The line [7-8] of listing 1.2 shows a call to a randomly initialized neural network model, the detail of which has been abstracted away. Upon each invocation of this sampling function, a neural network is randomly initialized, i.e., random initial biases and weights, and then the model is trained for the given hidden unit size. It finally returns the value of the loss function upon convergence of the neural network. In line 18 of listing 1.3, a hypothetical test is performed to determine if the error or loss distribution for current hidden unit size has sufficient samples in required confidence level, say 95%, within predefined acceptable loss range.

Therefore, unlike picking one random choice as random search does, we are now able to choose a hidden unit size, for which 95% of the randomly initialized neural network converged within desired loss range. Quantifying uncertainty, controlling conditionals and computations over uncertain aspects of the random search let developer choose a hyperparameter value which, in general, found to be more effective on a broader range of the population. The flexible interface of the Uncertain<T> data type makes it very convenient for the client end by hiding statistical detail, i.e., sampling, hypothetical test, in this regard.

4 Detailed Approach

This section discusses the proposed solution approach in detail. In particular, we explain what random search fails to achieve compared to our method, random search with uncertainty, and how hypothetical test is performed in order to obtain desired confidence level in selecting hyper-parameter value.

4.1 Why Uncertain Random Search

As discussed in previous sections, random search has been designed to save the exhaustiveness of grid search and find a best hyperparameter value with much less trials than grid search. To that end, random search tries a number of trials for a hyperparameter which are selected randomly. Often a random subset of original dataset or a representative subset, if possible, is chosen as input to the model for random search. A subset is chosen as the original dataset can be very large which may render repetitive trials of random search expensive or the entire population of the input domain might not be

available during design phase of the model. Therefore, grid search or random search attempts to approximate a best hyperparameter value over a random subset of the real input domain. A drawback with this approach is that it fails to generalize performance of a particular trial of a hyperparameter over large input domain. Consequently, failing to characterize the uncertainty in the input domain may lead to local optimal hyperparameter value which may perform poorly over real input domain.

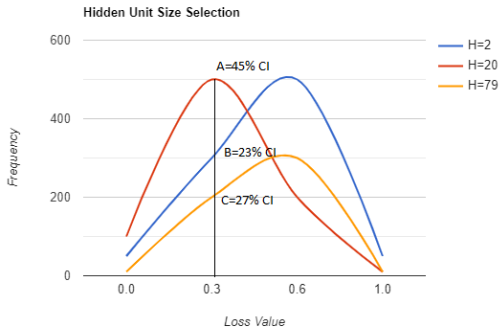


Fig. 2: Hidden unit size optimization with uncertain random search

Our proposed method, uncertain random search, takes this uncertainty of the input domain into account to optimize a hyperparameter with certain confidence level over uncertain input domain. For instance, consider the hidden unit size optimization example demonstrated in previous section. The line 6 of the listing 1.2 demonstrates how the uncertainty of large input domain is taken into consideration. Unlike random search which evaluates the model on a fixed subset, it initializes the model with a random subset of the original dataset on each invocation. Unlike evaluating a trial of hyperparameter on a fixed subset, uncertain random search forms a distribution of model performances for random subsets of input domain.

Fig. 2 demonstrates the outcome of uncertain random search for hidden unit size hyperparameter. We can see that for three random trials of the hyperparameter, H, it produces three loss function distribution. If we allow no worse than 0.3 as a loss value, then the distribution for 20 hidden unit size achieves maximum confidence interval (CI) i.e., 45% CI as indicated by point A on the graph. Therefore, uncertain random search picks 20 as a hidden unit size over other two trials since it has achieved best result over maximum subset of the population. However, an unaware random search might evaluate the model on a particular input subsets as uncertain random search does for H=2 or 79. As a result, it will pick other H-values than what uncertain version picks.

4.2 Hypothetical Test

Uncertain data type approximates distribution by random sampling which essentially introduces approximation error [5]. To reduce this approximation error, the framework

implicitly performs a hypothetical test. In hypothetical test, a null hypothesis is assumed to be true unless sufficient evidence at required confidence level for rejecting is found. In this section, we will briefly explain the hypothetical test for a hyperparameter optimization, hidden unit size.

The line 17 of listing 1.3 compares the loss distribution for a particular trial with a predefined threshold. If the loss distribution has sufficient samples that falls below the threshold at a certain confidence interval, then that trial is accepted. This procedure repeats until an optimal hidden unit size is found or maximum iteration is done. In this case, the null hypothesis, H_0, denotes the loss distribution having enough sample less or equal than the threshold at a certain confidence interval. On the other hand, the alternative hypothesis, H_A, states that at required confidence interval, there are not enough samples within acceptable loss range. The following steps are followed to perform this hypothetical test:

Step 1: k samples are drawn from the loss distribution.
Step 2: Check if H_A holds at the required confidence interval for the drawn samples.
Step 3: If the result is not significant, either draw another batch of k samples and repeat from step 2 with 2k samples now or terminate if maximum number of samples reached.

A higher confidence level would tend to select a hyperparameter value which is more closer to the global optimal as it will require more neural network models to agree on a certain hyperparameter value.

5 Experimental Design

In this section, we discuss our implementation module-wise in detail and outline the experimental setup we employed in this project.

First, we provide the implementation of DNN model used in the experiment. Then we introduce the details of Random Search and Uncertain<T> model. Finally, we have described the usage of our method in the client program and the evaluation setup.

5.1 DNN Model

In this section we introduce the details and parameters of the DNN model. we first briefly introduce our experimental data, because the DNN model is related to the input data shape. The data set we use is MNIST, which consists of $60,000$ training and $10,000$ testing pixel images of size 28×28. Since the input data is an image, two convolutional and one max-pooling layers are introduced to DNN model. It is followed by a flatten layer and a three layers of fully connected neural network, as shown in Fig. 3. The number of hidden units ($no\_of\_hidden\_unit$) in the hidden layer is the hyperparameter to be determined. The dropouts of the hidden layer and output layer are 0.25 and 0.5 respectively.

DNN model implements three basic functions, *create_model*, *train* and *evalute*. The *create_model* function gets hyperparameter set and generates a specific model with it.

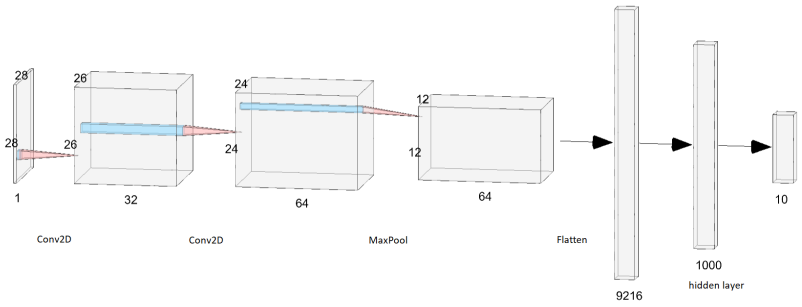


Fig. 3: DNN Model

The *train* function acquires training and testing data sets and trains the model. The *evaluate* function obtains the trained model and test data set, and returns the accuracy of the model. These three functions support the implementation of random search and uncertainty model experiment. We will show this in section 5.2 and 5.5. The programing language we use is *python3.7* with packages *keras, tensorflow, sklearn, etc*.

5.2 Random Search

Random search is proved to be an effective method to obtain hyperparameters. It includes following steps: First, generate the sampling space of the hyperparameter. Second, sample uniformly from hyperparameter space to get several hyperparameter set. Third, with each set, create, train and evaluate the model. Finally, choose the corresponding hyperparameter set of best performance model above. So we implement the random for DNN model as follows,

Step 1: Generate sample space for $no\_of\_hidden\_unit$, [1,1000].
Step 2: Sample n values uniformly from sample space, as $H = \{H_1, \cdots, H_n\}$.
Step 3: Create n corresponding models $\{M_j\}$ with $\{H_j\}$.
Step 4: Train and evaluate models $\{M_j\}$.
Step 5: Choose H_j for which M_j gives best results.

The steps shows above is one time of experiment. Since random search obtain best hyperparameter based on statistics, we need repeat the experiment many times. We choose to repeat it 13 time. By analysing the result we could get the conclusion. For data selection, we take 30% samples from origin train set as our training set. The set is used all over the experiments to guarantee the consistency. We use the entire origin test set as our test set. We aim to test the performance of random search when the training set is not sufficient. In addition, a smaller training set usually indicates fewer labeling tasks, which usually saves more labeling costs.

Table 2 shows the result of random search strategy. The first column shows the experiment id, which starts from 1 to 13. Second column shows the optimal hyperparameter, the number of hidden unit, the random search strategy choose from 10 trials random selection. The last column shows the accuracy of the model we create with the number of hidden unit from column 2.

ExperimentID	# of hidden unit	Accuracy
1	482	0.9556000232696533
2	436	0.9589999914169312
3	687	0.964900016784668
4	752	0.9707000255584717
5	603	0.9629999995231628
6	722	0.9643999934196472
7	989	0.9605000019073486
8	976	0.9664000272750854
9	415	0.9557999968528748
10	954	0.9351999759674072
11	560	0.9585000276565552
12	398	0.9545000195503235
13	388	0.9652000069618225

Table 2: Result of Random Search

5.3 Uncertain<T> Module

5.4 Uncertain Data Type

Inspired by the framework proposed in [5], we designed a first-order **Uncertain** data type to hold, sample, and perform hypothetical tests on the distributions.

```
class Uncertain:
    def __init__(self,sampler, *args):
        self.sampleSize=40
        self.samplingFunction = sampler
        self.args = list(args)

    def __lt__(self, other):
        return self.hypothesis_test(other, op.lt)


    def sample(self):
        return self.samplingFunction(*self.args)

    def hypothesis_test(self, other, H0):
        t1 = self.sample()
        t2 = other.sample()
        return H0(t1,t2)

    #returns sample mean and margin of error in 95% CI
    def E(self):
        data=[]
        for i in range(self.sampleSize):
            data.append(self.sample())
```

```
25          std=stat.stdev(data)
26          moe= (2*std)/math.sqrt(self.sampleSize)

28          return [stat.mean(data), moe]
```

Listing 1.4: Uncertain Data Type

Listing 1.4 shows a portion of our implementation of uncertain data type in Python. As we can see, <= comparison operator has been overloaded with method *__lt__* in line 7 to allow first-order operation in conditional on uncertain data-type. Similarly, other operators have been overloaded as well. Inside overloaded methods for comparison operators, a call to *hypothesis_test* function is performed to allow hypothetical testing.

The function *E* approximates population mean from a sample of size, *sampleSize*. In order to approximate population mean, first, *sampleSize* amount of samples are drawn from the distribution as shown in line 22 23. Then, standard deviation of the samples are computed in 25. Then, the margin of error (MOE) is computed for sample's mean in 95% confidence interval (CI) with following formula:

$$MOE = \frac{2s}{\sqrt{n}}$$

Here, *s* refers to sample standard deviation and n is the size of sample. Based on central limit theorem, this tells us with 95% CI that the actual population mean will lie within $[\bar{x} - MOE, \bar{x} + MOE]$ where $\bar{x}$ refers to sample mean.

5.5 Client Program

As discussed in 3, we create two *Uncertain* variable, one holding the distribution of hyper-parameter, hidden unit size, and another referring to neural network accuracy distribution.

```
2 def getRandomHiddenSizeSample():
3     return random.randint(1,1000)

5 _hiddenUnitSampler=Uncertain(getRandomHiddenSizeSample)

7 no_of_exp=10
8 no_of_trials=10

10 x_train, y_train, x_test, y_test, input_shape =
       input_data(30,100)

12 for j in range(no_of_exp):

14     result = []
```

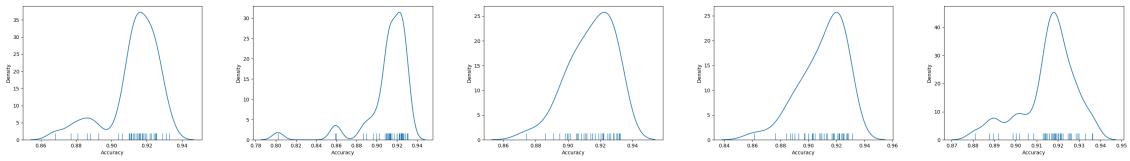
```
16    for i in range(no_of_trials):
17        hiddenUnitSize=_hiddenUnitSampler.sample()

19        _accuracySampler=Uncertain(train_and_evaluate,
    training_data_size_in_petrcent=10, epoch=1,
    hiddenUnitSize)

21        e=_accuracySampler.E()

23        result.append((hiddenUnitSize, e[0]-e[1]))


26    result=sorted(result, key=lambda x: x[1])

28    model = create_model(result[len(result)-1][0])
29    model = train(model, x_train, y_train, x_test, y_test
    , 1)
30    accuracy = evaluate(model, x_test, y_test)
```

Listing 1.5: Uncertain Data Type

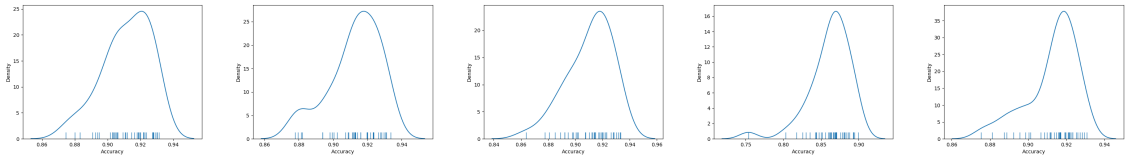
Listing 1.5 shows the implemenation of client program which is optimizing hidden unit size hyper-parameter of a neural network on *MNIST* dataset. In line 6, we create a variable to manipulate the distribution of possible hidden unit sizes where we pass the reference to a sampling function, *getRandomHiddenSizeSample*. As we can see, the sampling function for hidden unit distribution is randomly drawing a sample form the domain as random search does. In line 18, a sample is drawn from the distribution of hidden unit. Similarly, we define a *Uncertain* variable in line 20 for holding the distribution of network's accuracies. A reference to a network implementation is passed as a sampling function which returns accuracy of the network based on the parameters passed, including a hidden unit size.

Experiment Setup We have conducted 13 experiments in total where each experiment consists of 10 random trials. As we can see in line 17, for each experiment, *no_of_-trials* trials are performed. In each trial, a hidden unit size is sampled randomly which is shown in line 18. In the next line, the drawn hidden unit size is passed to a neural network implementation. Also, unlike random search, only small fraction of the dataset (10% in our case) is used for tuning.

In line 22, the expected accuracy is calculated in 95% CI. The expected accuracy refers to the population mean for current hidden unit size. Fig. 5 shows distribution of accuracies for some hidden unit size in our experiment. As an illustration, the expected accuracy for hidden unit size, 140, is 96% which lies around most congested area of Fig. 5(j). The benefit of this technique lies in the fact that when choosing one best trial, we don't rely on a single instance of accuracy which can be very low sometime or very high as can be seen in Fig. 5(j). Expected accuracy reduces this uncertainty due to random nature of the random search.

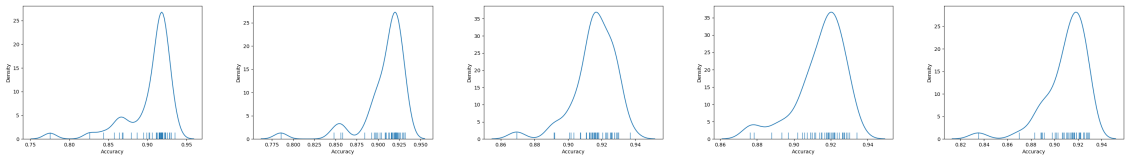


(a) H = 852 (b) H = 848 (c) H = 833 (d) H = 691 (e) H = 524

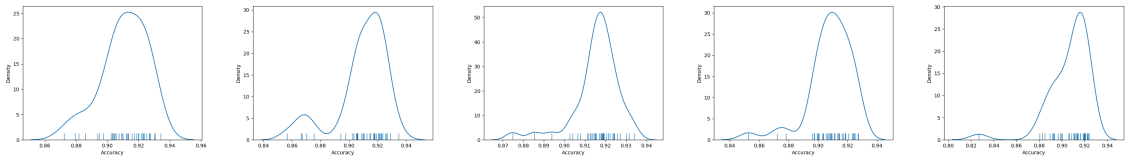


(f) H = 432 (g) H = 353 (h) H = 297 (i) H = 25 (j) H = 237

Fig. 4: Distributions for trials in experiment 1



(a) H = 947 (b) H = 933 (c) H = 869 (d) H = 838 (e) H = 403



(f) H = 380 (g) H = 326 (h) H = 290 (i) H = 252 (j) H = 140

Fig. 5: Distributions for trials in experiment 2

Finally, after completing all trails for a experiment, in line 27-31, the model is evaluated on whole dataset for the best trial and accuracy is recorded for the evaluation.

Similarly, we conducted 13 experiments with random search as well. Each experiment, in turn, conducted 10 trials. After choosing the best hyper-parameter in each experiment, the final accuracy value was computed on whole dataset and recorded for comparing with uncertain random search result.

6 Evaluation

In this section, we discuss the result of our methodology compared to random search technique on MNIST dataset [25]. MNIST dataset is a widely used dataset in many machine learning tasks. We have evaluated our method over two research questions:

- **RQ1:** How much uncertainty remains in hyperparameter optimization?
- **RQ2:** What is performance improvement when we optimize hyperparameter with our method over the random search method?

6.1 RQ1: How much uncertainty remains in hyperparameter optimization?

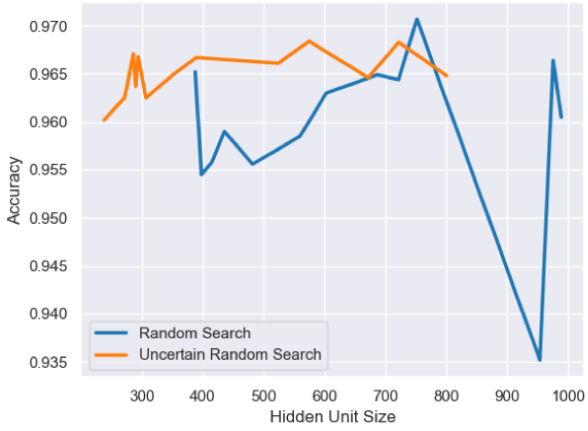


Fig. 6: Comparison of accuracy prediction between random search and uncertain search

To answer this research question, we have plotted the result of all 13 experiments for both our method and random search in a line chart as shown in Fig. 6. As we can see, the accuracy ranges from 96% to 97% in the case of uncertain random search, whereas, accuracy for random search ranges from 93% to 97%. However, random search is more uncertain as the accuracy fluctuates largely. On the other hand, uncertain random search provides consistent performance as it evaluates model accuracy on as many diverse population as possible. However, random search relies on only one sample from the accuracy distribution of a hidden parameter unlike our technique which draws as many samples as possible.

As we can see from the Fig. 5, accuracy may vary widely for a same hidden unit size. It is possible that, random search considers a trial to be best while it performs poorly on most of the samples. Eventually, it may lead to poor performance over unseen population which is illustrated by sharp drop in the curve of random search in Fig. 6. However, our technique doesn't rely on a single sample as explained in previous

section. Therefore, it provides a greater chance of better performance over the unseen population.

6.2 RQ2: What is performance improvement when we optimize hyperparameter with our method over the random search method?

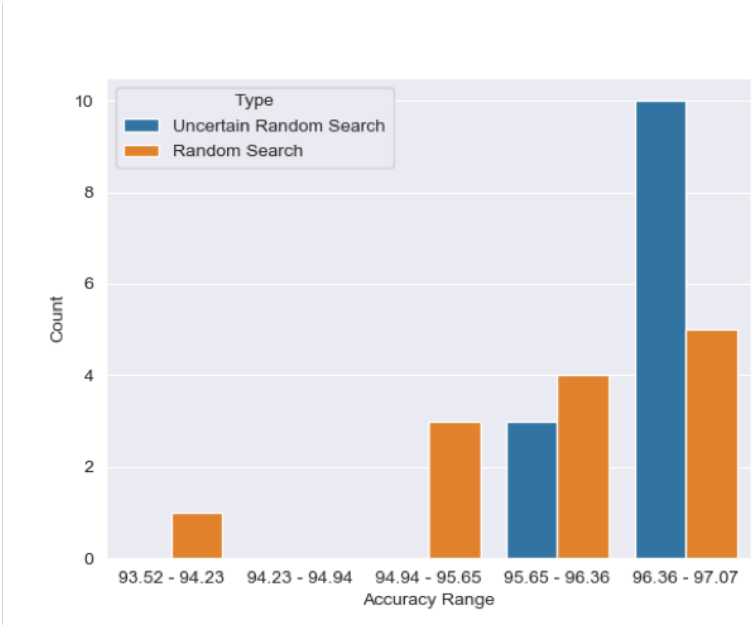


Fig. 7: performance comparison between random search and uncertain search

To answer this research question, we have plotted the result of all 13 experiments for both our method and random search in 5 bins of accuracies as shown in Fig. 7. As we can see, the accuracies from uncertain random search do not fall in the lower three bins unlike accuracies from random search.

Based on our result, there are 62% chance that that accuracy will be in lower ranges if random search applied compared to the 23% in uncertain random search. In other words, with uncertain version of random search, it is 77% likely that a randomly initialized model's accuracy will be in the high range. It can be attributed to the fact that, uncertain random search is better informed about a particular trial of a hyper-parameter. Unlike random search, it chooses the trial that best generalizes over the entire population which makes it more likely to produce better result.

7 Conclusion and Future Plan

In this project, we have discussed the presence and consequences of uncertainties in hyper-parameter optimization techniques, i.e. grid and random search specifically. We

further presented an idea to quantify these uncertainties to improve the overall performance of a hyper-parameter optimization technique, random search. To that end, inspired by Bornholt et. al.[5], we have designed a first order type in Python for uncertain data which provides a flexible interface for manipulating the distributions. Finally, we have evaluated our method, a uncertain version of random search, on MNIST dataset [25]. Our result indicates that taking uncertainty into account, while optimizing a hyper-parameter, improves overall performance significantly. In summary, uncertain random search produced more consistent accuracies as it evaluates model accuracy on as many diverse population as possible. Also, another important finding in our experiment is that, our technique requires less training data for tuning compared to the whole dataset needed by random search or grid search.

Inspired by the encouraging findings in this project, in future, we plan to evaluate our technique for unseen input domain with synthetic data generation algorithms. In addition, we plan to evaluate our technique on very large dataset, where typical hyper-parameter search techniques takes prohibitive amount of time, for faster tuning of the hyper-parameters.

References

1. Ghahramani, Z.: Bayesian non-parametrics and the probabilistic approach to modelling. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **371**(1984) (2013) 20110553
2. Jaynes, E.T.: Probability theory: The logic of science. Cambridge university press (2003)
3. Baudart, G., Hirzel, M., Mandel, L.: Deep probabilistic programming languages: A qualitative study. arXiv preprint arXiv:1804.06458 (2018)
4. Pfeffer, A.: Practical probabilistic programming. Manning Publications Co. (2016)
5. Bornholt, J., Mytkowicz, T., McKinley, K.S.: Uncertain<T>: A first-order type for uncertain data. In: ACM SIGPLAN Notices. Volume 49., ACM (2014) 51–66
6. Larochelle, H., Erhan, D., Courville, A., Bergstra, J., Bengio, Y.: An empirical evaluation of deep architectures on problems with many factors of variation. In: Proceedings of the 24th international conference on Machine learning, ACM (2007) 473–480
7. Hinton, G.E.: A practical guide to training restricted boltzmann machines. In: Neural networks: Tricks of the trade. Springer (2012) 599–619
8. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. Journal of Machine Learning Research **13**(Feb) (2012) 281–305
9. Falkner, S., Klein, A., Hutter, F.: Bohb: Robust and efficient hyperparameter optimization at scale. In: International Conference on Machine Learning. (2018) 1436–1445
10. Quiñonero Candela, J., Rasmussen, C.E., Sinz, F., Bousquet, O., Schölkopf, B.: Evaluating predictive uncertainty challenge. In: Proceedings of the First International Conference on Machine Learning Challenges: Evaluating Predictive Uncertainty Visual Object Classification, and Recognizing Textual Entailment. MLCW'05, Berlin, Heidelberg, Springer-Verlag (2006) 1–27
11. Zhang, S.: Knn-cf approach: Incorporating certainty factor to knn classification. IEEE Intelligent Informatics Bulletin **11** (2010) 24–33
12. Gal, Y., Ghahramani, Z.: Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48. ICML'16, JMLR.org (2016) 1050–1059

13. Gal, Y., Islam, R., Ghahramani, Z.: Deep bayesian active learning with image data. In: Proceedings of the 34th International Conference on Machine Learning - Volume 70. ICML'17, JMLR.org (2017) 1183–1192
14. Blundell, C., Cornebise, J., Kavukcuoglu, K., Wierstra, D.: Weight uncertainty in neural networks. In: Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37. ICML'15, JMLR.org (2015) 1613–1622
15. Lakshminarayanan, B., Pritzel, A., Blundell, C.: Simple and scalable predictive uncertainty estimation using deep ensembles. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. NIPS'17, USA, Curran Associates Inc. (2017) 6405–6416
16. Bornholt, J., Meng, N., Mytkowicz, T., McKinley, K.S.: Programming the internet of uncertain < t >hings. In: Sensors to Cloud Architectures Workshop (SCAW), ACM - Association for Computing Machinery (February 2015) 1–7
17. Boston, B., Sampson, A., Grossman, D., Ceze, L.: Probability type inference for flexible approximate programming. SIGPLAN Not. **50**(10) (October 2015) 470–487
18. Nandi, C., Grossman, D., Sampson, A., Mytkowicz, T., McKinley, K.S.: Debugging probabilistic programs. In: Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. MAPL 2017, New York, NY, USA, ACM (2017) 18–26
19. Ghahramani, Z.: Probabilistic machine learning and artificial intelligence. Nature **521** (05 2015) 452–9
20. Agarwal, S., Milner, H., Kleiner, A., Talwalkar, A., Jordan, M., Madden, S., Mozafari, B., Stoica, I.: Knowing when you're wrong: Building fast and reliable approximate query processing systems. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. SIGMOD '14, New York, NY, USA, ACM (2014) 481–492
21. Manousakis, I., Goiri, I.n., Bianchini, R., Rigo, S., Nguyen, T.D.: Uncertainty propagation in data processing systems. In: Proceedings of the ACM Symposium on Cloud Computing. SoCC '18, New York, NY, USA, ACM (2018) 95–106
22. Esfahani, N., Kouroshfar, E., Malek, S.: Taming uncertainty in self-adaptive software. In: SIGSOFT FSE. (2011)
23. Perez-Palacin, D., Mirandola, R.: Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation. (03 2014) 3–14
24. Géron, A.: Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems. " O'Reilly Media, Inc." (2017)
25. Lecun, Y.: The mnist database of handwritten digits