Extending Gaussian process emulation using cluster analysis and artificial neural networks to fit big training sets

DOI: 10.1080/17477778.2018.1489936
Handle: http://hdl.handle.net/1942/29854

Extending Gaussian process emulation using cluster analysis and artificial neural networks to fit big training sets

Wim De Mulder*†[1], Bernhard Rengs†[2], Geert Molenberghs[3,1], Thomas Fent[2] and Geert Verbeke[1,3]

[1]L-BioStat, KU Leuven, Belgium
[2]Wittgenstein Centre (IIASA, VID/ÖAW, WU), VID/ÖAW, Austria
[3]I-BioStat, University of Hasselt, Belgium

August 5, 2016

Abstract

Gaussian process emulation is a relatively recent statistical technique that provides a fast-running approximation to a complex computer model, given training data generated by the considered model. Despite its sound theoretical foundation, Gaussian process emulation falls short in practical applications where the training data set is very large, due to numerical instabilities in inverting the correlation matrix. We show how Gaussian process emulation can be extended to handle large training sets by first dividing the training set into smaller subsets using cluster analysis, then training an emulator for each subset, and finally combining the emulators using an artificial neural network. We furthermore compare the performance of multiple artificial neural network configurations with varying training parameters. Our work has also conceptual relevance, as it shows how to solve a big data problem by introducing a local level in input space, where each emulator specializes in a certain subregion, and a global level, where the identified local features of the computer model are combined into a global view.

Keywords: *Gaussian process emulation, artificial neural networks, cluster analysis, inverse distance weighting, agent-based models*

Acknowledgment

The authors acknowledge funding from the KU Leuven funded Geconcerteerde Onderzoeksacties (GOA) project 'New approaches to the social dynamics of long-term fertility change' [grant 2014–2018;GOA/14/001].

*E-mail: wim.demulder@cs.kuleuven.be, address: Kapucijnenvoer 35 blok d - bus 7001, 3000 Leuven, Belgium, phone: +32 16 37 33 24, fax: +32 16 3 37015.

†The first two authors contributed equally to this work.

1 Introduction

Gaussian process (GP) emulation (O'Hagan, 2006; Challenor, 2013; Kleijnen, 2014) has been developed to provide a fast-running approximation to a complex, slow-running computer model, given a training data set that has been generated by the computer model under consideration. GP emulation has several appealing characteristics: it has a rigorous mathematical foundation, it produces its output almost instantaneously (Conti and O'Hagan, 2010), and the validation and assessment of a GP emulator can be performed by sound statistical techniques (Bastos and O'Hagan, 2009). However, an important deficiency of the originally developed Gaussian process emulation method is that it cannot handle very large training sets. The reason is that the description of a GP emulator relies on the inverse of the correlation matrix having dimensions that are quadratic in the number of training data elements. Such an operation is extremely vulnerable to numerical instabilities.

The purpose of this paper is to show how GP emulation can be extended using techniques from the domains of cluster analysis and artificial neural networks, such that big training sets can be fitted. We demonstrate the applicability of our method using an agent-based model as case study, which generated a very large training data set containing over 10,000 training points. The empirical results are analyzed and compared to the performance of some simpler methods that are based on GP emulation and that we applied to the same training set in previous work.
Our method is also appealing from a conceptual point of view, as it uses a divide and conquer principle to consider the fitting problem at two different levels: a local level where local features of the computer model are identified, and a global level where the local features are integrated to provide a global view on the given model. Such an approach ensures intuitive understanding. At the same time, the described method is advanced, as it performs optimization of parameters at each level. Both levels are interconnected by using the outputs at the local level as inputs at the global level.

Our work is of high practical relevance, as we have entered the Big Data era, where the massive sample size and high dimensionality of Big Data introduce unique computational and statistical challenges (Fan et al., 2014). This does not only apply to the domain of agent-based models, where training data can be generated at almost no economical cost, but also to fields where data sets contain physical measurements. A prime example of such a field is molecular biology, where very big data sets have been collected, even though collecting measurements often comes at high financial costs. The reason for the existence of such massive molecular biology data sets is obviously that it can provide substantial benefits to society, for example to develop personalized medicine (Alyass et al., 2015). Our work might thus also be useful in other areas, such as biology, where researchers encounter the failure of methods that worked well in the past but are not accustomed to the big data sets that have come into existence very recently (Li and Chen, 2014).

2 Outline of the paper

In Section 3 we review the methods that play a role in our work. First, agent-based models, as we have recently developed an agent-based model which we wanted to approximate by a very fast-running surrogate model. Secondly, GP emulation that we originally intended to apply to the training set generated by our agent-based model, because of its appealing properties described above. Thirdly, cluster analysis, since it quickly turned out that GP emulation as it has been developed originally is not applicable to training sets of very large size such as ours, and a first intuitive step is then to subdivide the training set into smaller subsets

using cluster analysis and to train an emulator on each subset. Fourthly, inverse distance weighting which we used in previous work to combine the outputs of the several emulators into one approximation. Results of our previous work will be used as benchmark for the simulations obtained by the method described in this paper. Finally, artificial neural networks that we use here as a more advanced method than inverse distance weighting to aggregate the outputs of the individual emulators. In Section 4 we describe the agent-based model that we have previously developed and the training data set that was generated by our agent-based model. Section 5 reviews the Gaussian process emulation methods that we have previously developed and the empirical results that we obtained on the same training set. In Section 6 we describe how Gaussian process emulation can be extended using cluster analysis and artificial neural networks to be applicable to very large training sets. Empirical results are given in Section 7. Section 8 contains a short discussion of the relevance of our work, both from an empirical point of view and from a conceptual perspective.

3 Methodology

3.1 Gaussian process emulation

GP emulation provides an approximation to a mapping $\nu : \mathbb{R}^n \rightarrow \mathbb{R}^m$. In the remainder of the paper we will assume that $m = 1$, which applies to our case study described below. The approximation to ν, i.e. the emulator, is determined as follows. In the first step, it is assumed that nothing is known about ν. The value $\nu(\mathbf{x})$ for any $\mathbf{x}$ is then modeled as a Gaussian distribution with mean $m(\mathbf{x}) = \sum_{i=1}^{q} \beta_i \, h_i(\mathbf{x})$, where β_i are unknown coefficients and where h_i represent linear regression functions. The covariance between $\nu(\mathbf{x})$ and $\nu(\mathbf{x}')$ is modeled as $\mathrm{Cov}\Big(\nu(\mathbf{x}), \nu(\mathbf{x}') \,|\, \sigma^2\Big) = \sigma^2 \, c(\mathbf{x}, \mathbf{x}')$, where σ^2 denotes a constant variance parameter and where $c(\mathbf{x}, \mathbf{x}')$ denotes a function that models the correlation between $\nu(\mathbf{x})$ and $\nu(\mathbf{x}')$. We adopt the most common choice for c:

$$c(\mathbf{x}, \mathbf{x}') \quad = \quad \exp\Big[-\sum_i \Big((x_i - x_i')/\delta_i\Big)^2\Big] \tag{1}$$

with x_i and x_i' the ith component of $\mathbf{x}$ and $\mathbf{x}'$ resp., and where the δ_i represent parameters that can be optimized via maximum likelihood (Andrianakis and Challenor, 1999). In the second step, training data $(\mathbf{x}_1, \nu(\mathbf{x}_1)), \ldots, (\mathbf{x}_n, \nu(\mathbf{x}_n))$ are used to update the Gaussian distributions to Student's t-distributions via a Bayesian analysis. The mean of the Student's t-distribution in $\mathbf{x}$ is then considered the best approximation to $\nu(\mathbf{x})$. Therefore, we refer to this mean as $\hat{\nu}(\mathbf{x})$. It is given by

$$\hat{\nu}(\mathbf{x}) \quad = \quad m(\mathbf{x}) + U^T(\mathbf{x}) A^{-1} ([\nu(\mathbf{x}_1), \ldots, \nu(\mathbf{x}_n)]^T - H\boldsymbol{\beta})$$

with $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_q)^T$ and with H a matrix having $h_j(\mathbf{x}_i)$ as element on the ith row and the jth column. Furthermore, $U(\mathbf{x})$ contains the correlations, as given by (1), between $\mathbf{x}$ and each of the training data points $\mathbf{x}_i$, while A denotes the correlation matrix, containing the correlations between $\mathbf{x}_i$ and $\mathbf{x}_j$ for $i, j = 1, \ldots, n$. A crucial entity in the above expression is A^{-1}, the inverse of the correlation matrix. We found that the inversion operation for our very large training data set, described in Section 4 below, is computationally intractable. We resolved this by dividing the training data set into smaller subsets using cluster analysis (a methodology that is briefly reviewed in Section 3.4) and then training an emulator for each small subset. Approximations generated by these emulators are then combined into a final, unique approximation using an artificial neural network, as we describe in Section 6. Values for the β_i and for σ^2 can be determined

analytically (O'Hagan, 2006). The cited work is also a useful reference for a more detailed account on GP emulation.

3.2 Artificial neural networks

An artificial neural network (ANN) is a computational model that resembles the structure and working of the human brain, in the sense that complexity arises from the cooperation between otherwise simple units, also called neurons. In this paper we will restrict to feed-forward ANN, which contain no feedback loops. Such an ANN can be used to approximate a mapping $\nu : \mathbb{R}^n \to \mathbb{R}$, given training data $(\mathbf{x}_1, \nu(\mathbf{x}_1)), \ldots, (\mathbf{x}_n, \nu(\mathbf{x}_n))$. In the single-layer case, the approximation is modeled as

$$\hat{\nu}(\mathbf{x}) \quad = \quad g_2\left[\sum_{j=0}^{M} w_j^2 \, g_1\Big(\sum_{i=0}^{n} w_{ji}^1 x_i\Big)\right] \tag{2}$$

where the w_{ji}^1 and w_{kj}^2 are parameters, and where g_1 and g_2 are user-chosen so-called activation functions. A typical choice is the logistic activation function, given by $g(z) = 1/(1 + \mathrm{e}^{-z})$. An extra input component x_0 is introduced that has a fixed value $x_0 = 1$, to allow that the linear combination $\sum_{i=0}^{n} w_{ji}^1$ is shifted by an amount w_{j0}^1. A substantial number of algorithms exist to determine the parameters w_{ji}^1 and w_{kj}^2 as solutions to a sum-of-squares error function, a popular one being backpropagation (Bishop, 1996). The above formulation is referred to as the single-hidden-layer case, because there is only one set of parameters that is used in a linear combination of non input component values, namely the w_{kj}^2. We will also consider the two-hidden-layer case, which can be represented as

$$\hat{\nu}(\mathbf{x}) \quad = \quad g_3\left(\sum_{k=0}^{M_2} w_k^3 g_2\left[\sum_{j=0}^{M_1} w_{kj}^2 \, g_1\Big(\sum_{i=0}^{n} w_{ji}^1 x_i\Big)\right]\right) \tag{3}$$

Although heuristic methods exist to determine the number of parameters, given by M in the single-hidden-layer case (2) or by M_1 and M_2 in the two-hidden-layer case (3), see (Panchal and Panchal, 2014), it is good practice to try several values for M or M_1 and M_2, and to choose the number that results in the best performance according to a chosen evaluation criterion.

3.3 Inverse distance weighting

Inverse distance weighting (IDW) is an approximation method in a metric space setting, originally developed by Shepard in the context of spatial analysis and geographic information systems (Shepard, 1968). IDW approximates the unknown value of ν in a given point $\mathbf{x}$ as:

$$\hat{\nu}(\mathbf{x}) \quad = \quad \sum_{i=1}^{n} \frac{w_i(\mathbf{x})}{\sum_{j=1}^{n} w_j(\mathbf{x})} \nu(\mathbf{x}_i) \qquad \text{if } d(\mathbf{z}, \mathbf{z}_i) \neq 0 \ \forall i \tag{4}$$

$$\quad = \quad \nu(\mathbf{x}_i) \qquad\qquad\qquad\qquad \text{otherwise} \tag{5}$$

with $w_i(\mathbf{x}) = 1/d(\mathbf{x}, \mathbf{x}_i)^\alpha$, where d is any metric and where α is a constant larger than zero.

3.4 Cluster analysis

Cluster analysis is the unsupervised partitioning of a data set into groups, also called clusters, such that data elements that are member of the same group have a higher similarity than data elements that are member of different groups (Jain et al., 1999). Similarity is typically expressed in terms of a user-defined distance measure, such as the commonly used Euclidean distance. Arguably the best known clustering algorithm is k-means (Jain et al., 1999), an iterative algorithm that not only determines clusters but also centers for the obtained clusters.

In previous work, we have applied k-means for two purposes. First, we intended to use GP emulation to approximate our ABM, but due to the very large size of our training data set and the related instabilities in inverting the correlation matrix, direct application of GP emulation turned out to be infeasible. Thus we applied cluster analysis, thereby creating smaller training data sets, each of them being a subset of the original training data set. A separate emulator was then trained with each of these small data sets. Secondly, the number of clusters that was determined with k-means gave at once a suitable number of basis functions in our RBFN and the cluster centers were used as the centers of the basis functions. The results of the performed cluster analysis are outlined below.

3.5 Agent-based models

An agent-based model (ABM) is a computational model that simulates the behavior and interactions of a multitude of heterogeneous autonomous agents. A key feature is that population level phenomena are studied by explicitly modeling the interactions of the individuals in these populations (Gilbert, 2007; Macal and North, 2010; Macal, 2016). The systems that emerge from such interactions are often complex and might show regularities that were not expected by researchers in the field who solely relied on their background knowledge about the characteristics of the lower-level entities to make predictions about the higher-level phenomena. ABMs are especially popular among sociologists who model social life as interactions among heterogeneous adaptive agents who influence one another in response to the influence they receive (Billari and Prskawetz, 2003; Billari et al., 2006).

4 Description of our agent-based model and the training data set

The decision about having children has manyfold impacts on the individual life course and, consequently, diverse issues are taken into consideration. Besides the economic situation, economic insecurity (Modena et al., 2014), family policies and gender policies (Baizan et al., 2016), childbearing decisions are usually taken under the influence of friends and relatives (Bernardi and Klärner, 2014; Balbo and Barban, 2014). Since ABMs offer a wide range of possibilities to consider the influence of social networks on individual decisions (Klabunde and Willekens, 2016) the process of taking a childbearing decision and actually giving birth is well-suited for applying ABMs.

In previous work, we developed an agent-based model to analyze the effectiveness of family policies under different assumptions regarding the social structure of a society (Fent et al., 2013). The agents represent the female partner in a household. They are heterogeneous with respect to age, household budget, parity, and intended fertility. A network of mutual links connects the agents to a small subset of the population to exchange fertility preferences. The agents are endowed with a certain amount of time and money which they

allocate to satisfy their own and their children's needs. We considered two components of family policies: 1. the policy maker provides a fixed amount of money or monetary equivalent per child to each household and 2. a monetary or non-monetary benefit proportional to the household income is received by the household. The output on the aggregate level that is produced by the ABM consists of the cohort fertility, the intended fertility and the fertility gap. The inputs include the level of fixed and income dependent family allowances, denoted by b^f and b^v, and parameters that determine the social structure of a society, such as a measure for the agents' level of homophily α, and the strength of positive and negative social influence, denoted by pr_3 and pr_4 resp. The results of the application of our ABM and the related sociological findings can be found in the cited work.

We generated a training data set by applying our ABM to varying values for the input variables b^f, b^v, α, pr_3 and p_4, a selection of the larger amount of variables in our ABM. These five variables were found to have the largest influence on the outcomes. On the output side we restrict attention to one variable, namely cohort fertility. The ABM was applied to 10,732 vectors in the input domain, which gives a very large training data set. A test data set containing 500 input-output pairs was generated to evaluate the performance of several approximation methods.

5 Previous work

In previous work, we have applied several approximation methods to the training data set described above (De Mulder et al., 2015). We will use the results of these applications to evaluate the performance of our ANN based method, described in Section 6. To make this paper self-contained, we briefly review the methods that were applied and the results that were obtained.

5.1 Previous methods

Originally we intended to use GP emulation to fit the generated training data set. However, due to its very large size this was impossible, since the inverse of the correlation matrix is a crucial entity in the formulation of the approximation and it was intractable to compute this inverse (in fact, it was even impossible to store the correlation matrix in computer memory). Therefore, we applied the k-means cluster analysis algorithm (Jain et al., 1999) to subdivide the training set into 34 subsets and we trained a separate emulator on each subset. Our previous work showed that with this number of clusters there is an appropriate balance between the desire to avoid numerical instabilities in inverting the correlation matrices (implying a large number of clusters) and the desire to have a large number of points in each subset to provide each emulator with enough training examples (implying a small number of clusters). The implemented methods then differ in the way the outputs of the emulators are combined into one final output.

The first six methods only use the outputs of the k closest emulators with $k \in \{1, 2, 3\}$. The term *closest* requires the definition of a metric. We consider two metrics: the distance $d(\mathbf{x}, E_i)$ from an input point $\mathbf{x}$ to the ith emulator is either defined as the Euclidean distance from $\mathbf{x}$ to the center of the ith cluster or as the minimum of all Euclidean distances from $\mathbf{x}$ to each of the training data points belonging to the ith cluster. We refer to the first metric as the center metric and to the second one as the minimum metric. The outputs are then combined using IDW as follows:

$$\hat{\nu}(\mathbf{x}) = \sum_{i=1}^{k} \frac{1/d(\mathbf{x}, E_i)}{\sum_{j=1}^{n} 1/d(\mathbf{x}, E_j)} \hat{\nu}_{E_i}(\mathbf{x}) \quad \text{if } d(\mathbf{x}, E_i) \neq 0 \ \forall i \tag{6}$$

$$= \hat{\nu}_{E_i}(\mathbf{x}) \quad \text{otherwise} \tag{7}$$

with $E_1, \ldots, E_k$ the k closest emulators to the given input point $\mathbf{x}$ and where $\hat{\nu}_{E_i}(\mathbf{x})$ refers to the approximation of the output in $\mathbf{x}$ provided by the ith closest emulator, as given by (2). Since $k \in \{1, 2, 3\}$ and since d either refers to the center metric or the minimum metric, this formulation indeed results in six possible implementations.

We also implemented a method where the output of each emulator is taken into account. The formulation is the same as (6)-(7) with $k = 34$ and where we have chosen d as the center metric.

5.2 Previous results

The seven methods described above are evaluated using the test data set. Given a test input point $\mathbf{x}$ with corresponding true output $\nu(\mathbf{x})$ and an approximation $\hat{\nu}(\mathbf{x})$ produced by one of the seven methods, we can evaluate the quality of the approximation as the relative difference between $\nu(\mathbf{x})$ and $\hat{\nu}(\mathbf{x})$ as follows:

$$RD(\mathbf{x}) = \left| \frac{\hat{\nu}(\mathbf{x}) - \nu(\mathbf{x})}{1/2(\hat{\nu}(\mathbf{x}) + \nu(\mathbf{x}))} \right|$$

The average relative difference for a particular method, denoted ARD, is then the average of $RD(\mathbf{x})$ over all test points $\mathbf{x}$. The results we obtained for the seven methods are given in Table 1.

Table 1: Previous methods and results

Method	**Description**	**ARD**
E_{1_min}	closest emulator, minimum distance	0.14
E_{1_cen}	closest emulator, center distance	0.17
E_{2_min}	2 closest emulators, minimum metric	0.21
E_{2_cen}	2 closest emulators, center metric	0.23
E_{3_min}	3 closest emulators, minimum metric	0.24
E_{3_cen}	3 closest emulators, center metric	0.26
E_{34_cen}	all emulators, center metric	0.29

6 Description of ANN based method

The basic principles of all previous methods are similar: split the training data set into subsets using cluster analysis, apply GP emulation to each subset, and aggregate the produced outputs into a final output using IDW. IDW is a simple method that approximates an unknown output as a convex combination of known outputs. Its simplicity follows mainly from the fact that no optimization principle is involved. A research question of high practical relevance is whether a more complex aggregation of the outputs, taking into

account an (in all probability) increase in computation time, can result in a substantial closer fit. To answer this research question, we employ ANN, a method that is certainly much more advanced than IDW.

We perform simulations with many ANN architectures, single hidden layer as well as double hidden layer. The training data set for each ANN consists of elements of the following form:

$$\Big((\hat{\nu}_{E_1}(\mathbf{x}_j), \ldots, \hat{\nu}_{E_{34}}(\mathbf{x}_j)), \nu(\mathbf{x}_j)\Big)$$

where $\mathbf{x}_j$ is the input point of the jth element from the training data set generated by the ABM, where $\hat{\nu}_{E_i}(\mathbf{x}_j)$ refers to the approximation of the ith emulator for the output in $\mathbf{x}_j$ and with $\nu(\mathbf{x}_j)$ the output of our ABM in $\mathbf{x}_j$. In other words, the outputs of the emulators are given as inputs to an ANN and the single output of the latter is then used as approximation to the output in $\mathbf{x}$.
As algorithm to determine the parameters we used resilient backpropagation with weight backtracking (Riedmiller and Braun H, 1993). The activation function was chosen as the logistic function. Different architectures are then obtained by varying the threshold θ (i.e. a numeric value specifying the threshold for the partial derivatives of the error function as stopping criteria), the number of hidden layers (i.e. single-hidden-layer or double-hidden-layer) and the number of hidden neurons (i.e. either M in the single-hidden-layer case, or M_1 and M_2 in the double-hidden-layer case). We compare architectures with even and odd numbers of hidden neurons ranging from very small numbers up to two thirds of the number of inputs; ($M \in \{5, 8, 11, 14, 17, 20, 23\}$) in the one -hidden-layer case and $(M_1, M_2) \in \{(3, 2), (5, 3), (7, 4), (9, 5), (11, 6), (13, 7), (15, 8)\}$ in the two-hidden-layer case. Furthermore, we compare a number of different thresholds starting at $\theta = 0.05$ increasing in steps of 0.025 up to $\theta = 0.5$, then increasing with bigger steps of 0.125 up to $\theta = 5$. We trained and evaluated more ANNs in the former range of θ, as preliminary calculations showed that approx. computation time varies greatly it.

Implementation was performed in R with the neuralnet package (Günther and Fritsch, 2010). For each architecture 10 repetitions were performed, meaning that 10 ANNs were trained with the specified architecture for different values of the parameters. In total we implemented 770 different architectures, although for 17 of them none of the 10 repetitions converged. For the other architectures, in case that multiple repetitions converged successfully, we selected that repetition, which had the highest actual error (reached threshold) so that results would as closely as possible reflect performance of the associated θ. We thus evaluated 753 of 7280 successfully trained ANNs, each having a different architecture. A table describing the different architectures with information on approximate training duration, mean training steps, how many repetitions converged as well as performance, can be found in the Extended Appendix and in condensed form in the Appendix.

7 Results

To summarize the results found in the Appendix, Fig. 1 shows a histogram of the ARDs. As training the networks involved random initial values for the connection weights, training progress and predictive performance of networks with very similar or even identical architecture is stochastic and may vary to some degree. The histogram nevertheless clearly shows that the ANN based method performs much better than the IDW based methods that were previously implemented. Furthermore, only a single ANN's ARD was higher than the ARD of the best IDW based result (0.14) while the ARD of 99% of all evaluated ANNs was below 0.12, while the vast majority of ANNs of all architectures performed much better. The best ARD was 0.08, while the mean was 0.093, with the median being even slightly better at 0.092.

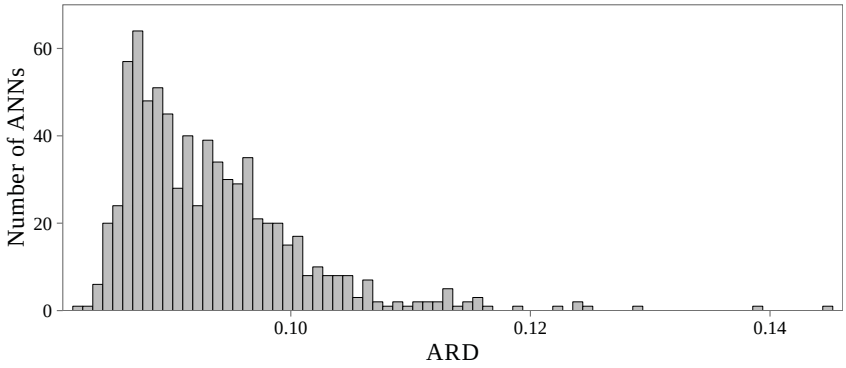

Figure 1: ARD of the ANN based methods

7.1 Computation Time of ANN based methods

As mentioned before, we consider ANN methodology to be more advanced and computationally intensive than other methods, e.g. IDW. As the potentially very time consuming part of employing ANNs is training the network, we investigated the influence of different architectures on training time and predictive performance. For a given maximum possible number of training steps ($1e+05$), repetitions (10), algorithm, error and activation functions, the chosen threshold θ obviously has a very strong impact on convergence and computation time. As can be seen in Fig. 2, approximate computation time (given in days) is very high for very low θ, but decreases quickly with higher values of θ. Furthermore, when comparing training time of architectures with the same θ value, ANNs with two hidden layers converge faster and fail to converge less often than ANNs with one hidden layer (93% of one hidden layer ANNs and 96% of the two hidden layer ANNs converged. For architectures with $\theta > 0.1$ at least one repetition of every parameter combination converged, whereas for smaller values the two-hidden-layer configurations converged more often than the one-hidden-layer configurations. For $\theta = 0.05$ none and for $\theta = 0.075$ only 3% of the repetitions of single-hidden-layer configurations converged, whereas 31% and 53% respectively of the repetitions of two-hidden-layer configurations converged. Preliminary calculations showed that ANNs with $\theta < 0.05$ would not converge with our data, even when increasing the maximum possible number of training steps ($1e+06$, $1e+07$) or increasing the number of hidden neurons.

Furthermore, 96.4% of ANNs with two hidden layers needed less computation time (on average only 54.1% of the time) than ANNs with the same total number of hidden neurons, which were arranged in one hidden layer but otherwise trained with the same parameters. One reason that ANNs with two hidden layers need less computation time than those with a single hidden layer with the same number of hidden nodes (when the number of hidden neurons is smaller than the number of inputs) is that they have less connections and thus weights to be trained. The other reason is that those with two hidden layers on average need fewer training steps for most configurations as can be seen in Table 2 (standard deviation is high due to the training threshold and partially due to initial random values of weights). The exception are ANNs with very low numbers of hidden neurons, i.e. M=5 and (M_1=3,M_2=2) where the two hidden layer configurations need more steps to converge than the one hidden layer configuration, as they seem to be almost too simple to cope with the complexity of our data (we could not successfully train ANNs with fewer hidden neurons in preliminary calculations. This seems to be confirmed by the fact that the ratio of convergence of the

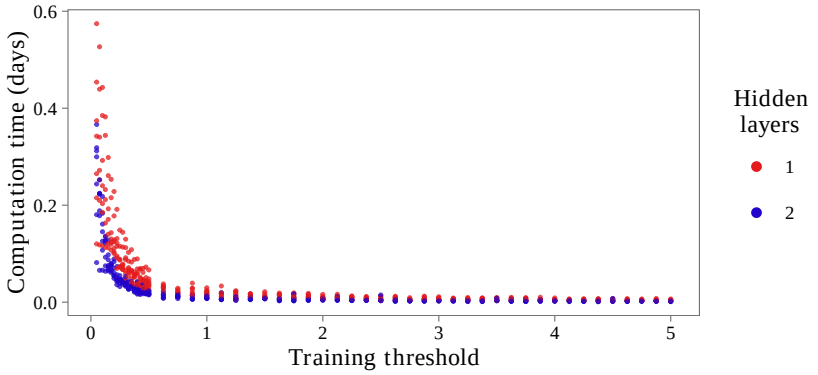

Figure 2: Computation time of different ANN architectures by number of hidden layers

(M_1=3,M_2=2) configuration is still below 100% for very high thresholds (up to $\theta = 4.5$), whereas all other architectures converged in 100% of our training cases with $\theta > 1$.

Table 2: Necessary training steps (in thousands) of ANNs

Hidden layers	**Number of hidden neurons**						
	5	**8**	**11**	**14**	**17**	**20**	**23**
1	25.7	21.8	18.0	16.2	16.7	15.7	16.1
	(32.2)	(30.3)	(26.7)	(24.4)	(26.8)	(25.2)	(25.6)
2	29.1	14.9	10.9	8.7	7.4	6.9	6.6
	(32.7)	(23.9)	(19.8)	(16.9)	(15.4)	(14.0)	(14.8)

In Table 3 we can see that ANNs with high numbers of hidden neurons take longer to train than those with lower numbers of neurons. Even though there are less steps needed, there are more weights to be trained in each step.

Table 3: Approximate training duration (in days) of ANNs

Hidden layers	**Number of hidden neurons**						
	5	**8**	**11**	**14**	**17**	**20**	**23**
1	0.032	0.038	0.047	0.042	0.067	0.051	0.081
	(0.039)	(0.056)	(0.069)	(0.067)	(0.111)	(0.083)	(0.133)
2	0.021	0.025	0.026	0.026	0.027	0.031	0.031
	(0.022)	(0.041)	(0.049)	(0.051)	(0.055)	(0.062)	(0.063)

7.2 Performance of ANN based methods

As can be seen in Fig. 3 performance of ANNs was lower (higher ARD) for those architectures with higher thresholds. Interestingly, we did not find an indication for overfitting of ANNs trained with very low

thresholds as these performed similarly well than those with higher thresholds. As indicated before, ANNs with lower thresholds, that might be prone to overfitting would not converge during preliminary calculations in the first place. Furthermore, even though performance decreased for higher thresholds, computation time decreased much quicker than performance decreased.

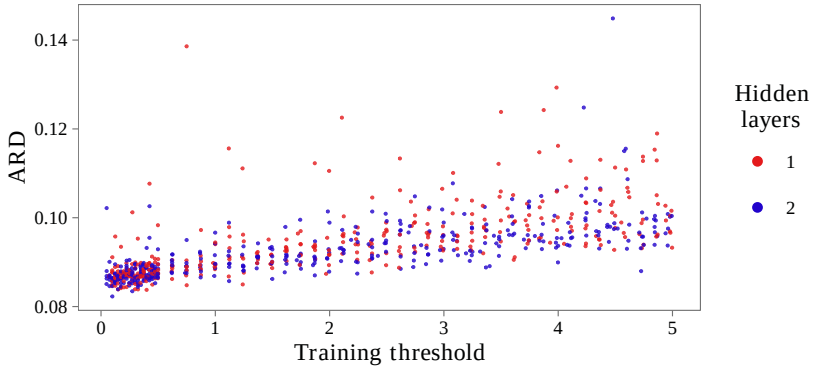

Figure 3: ARD of different ANN architectures by number of hidden layers

As shown before, ANNs with two hidden layers needed considerably less computation time, while their performance was comparable. Table 4 shows that ANNs with two hidden layers performed better mainly for higher numbers of hidden neurons. Overall, two-hidden-layer ANNs performed better in 60.2% of the cases, but only by a very small margin, as their ARD was on average 98.4% of single hidden layer configurations of the same number of hidden neurons. Comparing the performance of ANNs with regards to the number of hidden neurons we can see that they on average performed very similar, especially the one-hidden-layer results are almost identical.

Table 4: Training Configuration and Evaluation Results of ANNs

Hidden layers	**Sum of hidden neurons**						
	5	**8**	**11**	**14**	**17**	**20**	**23**
1	0.095	0.093	0.094	0.095	0.094	0.095	0.094
	(0.009)	(0.005)	(0.007)	(0.011)	(0.007)	(0.009)	(0.008)
2	0.094	0.094	0.093	0.091	0.092	0.091	0.092
	(0.006)	(0.005)	(0.007)	(0.005)	(0.008)	(0.005)	(0.006)

In addition to the performance of different ANNs with regards to different ranges of thresholds Table 5 shows the approximate computation time in italics for both one-hidden-layer and two hidden-layer configurations. As can be seen when moving from the range of $0.05 \leq \theta \leq 0.5$ to the range of $0.5 < \theta \leq 1$ for one-hidden-layer ANNs, mean ARD only slightly increases by 3.4% while mean computation time decreases by 81.4%. Similarly for two-hidden-layer ANNs, mean ARD on average increases by the same value, while computation time decreases by 83.8%.

We can conclude that by training ANNs at rather high thresholds combined with -in comparison to the number of inputs rather simple hidden-neuron configurations -we can reduce computation costs of training ANNs to feasible levels (ultimately minutes instead of hours). At the same time, it seems to be possible to

Table 5: Training configuration, evaluation results and computation time of ANNs

Hidden layers	**Threshold**									
	$\leq$ **0.5**	$\leq$ **1**	$\leq$ **1.5**	$\leq$ **2**	$\leq$ **2.5**	$\leq$ **3**	$\leq$ **3.5**	$\leq$ **4**	$\leq$ **4.5**	$\leq$ **5**
1 h.l. (ARD)	0.088	0.091	0.093	0.094	0.096	0.097	0.100	0.102	0.101	0.102
1 h.l. (Dur)	*0.129*	*0.024*	*0.015*	*0.012*	*0.009*	*0.007*	*0.006*	*0.006*	*0.005*	*0.004*
2 h.l. (ARD)	0.087	0.090	0.091	0.092	0.094	0.095	0.095	0.097	0.101	0.099
2 h.l. (Dur)	*0.068*	*0.011*	*0.008*	0.006	*0.005*	*0.003*	*0.003*	*0.003*	*0.003*	*0.002*

retain comparably good results, which are still noticeably better than the other methods which we evaluated in previous work.

8 Discussion

The presented simulation results clearly show that the aggregation of the outputs of the individual emulators using an ANN is superior to combining these outputs via IDW. Although IDW is conceptually and computationally much simpler than an ANN, this result should not be considered trivial. Notice that in IDW the coefficients in the convex combination of the known outputs, see (6)-(7), are a function of the given input point, while the parameters of the ANN based method, see (2) and (3), which also play the role of coefficients in a linear combination of certain values, are constant. Thus IDW tries to produce good approximations by using flexible coefficients while keeping the mapping itself simple. The opposite is true for artificial neural networks. In this sense, our work indicates the utmost importance of choosing a mapping that is complex enough to fit the given training data set.

Our work is also interesting from a conceptual point of view. We have shown how a computer model, in this case our ABM, can be approximated by considering two levels. First, a local level consisting of smaller regions in input space that are identified using cluster analysis. For each such region an emulator is trained using the subset of the training data set that lies in this region. Thus each emulator specializes in a certain part of the computer model, while being unaware of the remaining part. Secondly, a global level where the emulators are combined using an ANN, such that a global view on the computer model is obtained. At each level an optimization step is involved: at the local level the parameters of the emulators are optimized using the training data set, and at the global level the parameters of the ANN are optimized using the same training outputs but having as corresponding inputs the outputs of the emulators.
That is, approximating a computer model given a big training data set generated by it, can be done by a divide and conquer approach: split the problem into smaller problems, solve each of the smaller problems, and solve as remaining problem the aggregation of the solutions in such a way that a good overall approximation is obtained. Our ANN based method optimizes the solutions of the subproblems as well as of the remaining global problem.

9 Conclusion and future work

In this paper we have shown how a very big training set can be fitted. Broadly speaking, our method identifies subregions in input space using cluster analysis, then constructs an GP emulator for each subregion

and combines these emulators into a final approximator using an ANN. Furthermore we have shown that even quite simple ANN configurations can yield comparably good results as more complex ones, while having the advantage of rendering the ANN approach computationally feasible.

Our work opens the door to at least two interesting research questions that are very relevant for practical applications:

- In our study we applied GP emulation at the local level and an ANN at the global level. However, there is nothing that prevents the use of other methods at either level, e.g. ANNs at the local level *and* an ANN at the global level. An empirical study of varying combinations of methods would be very useful to answer important questions such as: 'do certain methods systematically collaborate better than other methods?'
- The methods at the local and the global level are applied one after the other, and at both levels there is an optimization step involved. This raises the question as whether a bad model fit at one of the levels can be compensated at the other level. This can be easily analyzed by introducing noise at one of the levels and evaluate how this influences the overall results.

References

[1] Alyass A, Turcotte M and Meyre D (2015). From big data analysis to personalized medicine for all: challenges and opportunities. *BMC Medical Genomics* **8**:33.

[2] Andrianakis I and Challenor P G (1999). The effect of the nugget on Gaussian process emulators of computer models. *Computational Statistics and Data Analysis* **35**(4): 1563-1581.

[3] Balbo N and Barban N (2014). Does fertility behaviour spread among friends? *American Sociological Review* **79**(3): 412-431.

[4] Baizan P, Arpino B and Delclòs C E (2016). The effect of gender policies on fertility: The moderating role of education and normative context. *European Journal of Population* **32**: 1-30.

[5] Bastos L S and O'Hagan A (2009). Diagnostics for Gaussian process emulators. *Technometrics* **51**(4): 425-438.

[6] Bernardi L and Klärner A (2014). Social networks and fertility. *Demographic Research* **30**: 641-670.

[7] Billari F C and Prskawetz, A (2003). Agent-based computational demography: Using simulation to improve our understanding of demographic behaviour. Heidelberg, Germany: Physica-Verlag.

[8] Billari F C, Fent T, Prskawetz, A and Scheffran J (2006). Agent-based computational modelling: Applications in demography, social, economic and environmental sciences. Heidelberg, Germany: Physica-Verlag.

[9] Bishop C (1996). Neural networks for pattern recognition. Clarendon Press.

[10] Challenor P (2013). Experimental design for the validation of kriging metamodels in computer experiments. *Journal of Simulation* **7**(4): 290-296.

[11] Conti S and O'Hagan A (2010). Bayesian emulation of complex multi-output and dynamic computer models. *Journal of Statistical Planning and Inference* **140**(3): 640-651.

[12] De Mulder W, Rengs B, Molenberghs G, Fent T and Verbeke G (2015). Statistical Emulation Applied to a Very Large Data Set Generated by an Agent-based Model. In: *Proceedings of the Seventh International Conference on Advances in System Simulation* 43-48.

[13] Fan J, Han F and Liu H (2014). Challenges of big data analysis. *National Science Review* **1**(2): 293-314.

[14] Fent T, Aparicio Diaz B and Prskawetz A (2013). Family policies in the context of low fertility and social structure. *Demographic Research* **29**: 963-998.

[15] Gilbert N (2007). Agent-based models: quantitative applications in the social sciences. SAGE Publications, Inc.

[16] Günther F and Fritsch S (2010). Neuralnet: Training of Neural Networks. *The R Journal* **2**(1): 30-38.

[17] O'Hagan A (2006). Bayesian analysis of computer code outputs: A tutorial. *Reliability Engineering & System Safety* **91**(6): 1290-1300.

[18] Jain A K, Murty M N and Flynn, P J (1999). Data clustering: a review. *ACM Computing Surveys* **31**: 264-323.

[19] Klabunde A and Willekens F (2016). Decision-making in agent-based models of migration: State of the art and challenges. *European Journal of Population* **32**(1): 73–97.

[20] Kleijnen J P C (2014). Simulation-optimization via Kriging and bootstrapping: a survey. *Journal of Simulation* **8**(4): 241-250.

[21] Li Y and Chen L (2014). Big biological data: challenges and opportunities. *Genomics, Proteomics & Bioinformatics* **12**(5): 187-189.

[22] Macal M C (2016). Everything you need to know about agent-based modelling and simulation. *Journal of Simulation* **10**(2): 144–156.

[23] Macal M C and North M J (2010). Tutorial on agent-based modelling and simulation. *Journal of Simulation* **4**(3): 151–162.

[24] Modena F, Rondinelli C and Sabatini F (2014). Economic insecurity and fertility intentions: The case of Italy. *Review of Income and Wealth* **60**: S233-S255

[25] Panchal F S and Panchal M (2014). Review on methods of selecting number of hidden nodes in artificial neural network. *International Journal of Computer Science and Mobile Computing* **3**(11): 455-464.

[26] Riedmiller M and Braun H (1993). A direct adaptive method for faster backpropagation learning: the RPROP algorithm . In: *IEEE International Conference on Neural Networks* 586-591.

[27] Shepard D (1968). A two-dimensional interpolation function for irregularly-spaced data. In: *Proceedings of the 1968 ACM National Conference*.

Appendix

Table 6 presents evaluation results for single hidden layer ANN's. Values in square brackets represent the minimum and maximum values for all of the following hidden neuron configurations [5,8,11,14,17,20,23]. The column "approximate duration" reflects computation time for attempting to train 10 repetitions of the given architecture in days.

Table 6: Evaluation results of ANNs with one hidden layer

Threshold	Hid. Neuron	ARD	Approx. dur.	Mean Steps	# of ANNs
0.05	[5,23]	[NA,NA]	[0.13,0.58]	[100000,100000]	[0,0]
0.075	[5,23]	[NA,NA]	[0.12,0.53]	[97859,100000]	[0,1]
0.1	[5,23]	[0.085,0.091]	[0.12,0.45]	[83685,99169]	[1,7]
0.125	[5,23]	[0.085,0.096]	[0.12,0.39]	[59416,94917]	[3,9]
0.15	[5,23]	[0.084,0.089]	[0.12,0.3]	[58099,90207]	[5,10]
0.175	[5,23]	[0.084,0.093]	[0.11,0.26]	[43676,91828]	[6,10]
0.2	[5,23]	[0.084,0.088]	[0.11,0.23]	[40782,89255]	[4,10]
0.225	[5,23]	[0.084,0.09]	[0.07,0.2]	[33524,71425]	[7,10]
0.25	[5,23]	[0.085,0.088]	[0.08,0.15]	[27857,61265]	[9,10]
0.275	[5,23]	[0.086,0.101]	[0.07,0.15]	[25166,56103]	[8,10]
0.3	[5,23]	[0.084,0.09]	[0.05,0.13]	[24919,41202]	[9,10]
0.325	[5,23]	[0.084,0.095]	[0.06,0.11]	[19202,59006]	[7,10]
0.35	[5,23]	[0.085,0.09]	[0.05,0.11]	[18969,51007]	[8,10]
0.375	[5,23]	[0.087,0.089]	[0.04,0.09]	[16330,49161]	[9,10]
0.4	[5,23]	[0.084,0.09]	[0.04,0.09]	[17463,29452]	[10,10]
0.425	[5,23]	[0.086,0.108]	[0.04,0.07]	[13699,30601]	[10,10]
0.45	[5,23]	[0.085,0.091]	[0.03,0.07]	[13835,35458]	[9,10]
0.475	[5,23]	[0.086,0.091]	[0.03,0.08]	[10605,27032]	[10,10]
0.5	[5,23]	[0.086,0.098]	[0.03,0.07]	[11334,26610]	[10,10]
0.625	[5,23]	[0.086,0.09]	[0.03,0.04]	[9253,20443]	[10,10]
0.75	[5,23]	[0.085,0.139]	[0.02,0.03]	[7449,14370]	[10,10]
0.875	[5,23]	[0.087,0.097]	[0.02,0.04]	[6166,19581]	[10,10]
1	[5,23]	[0.088,0.094]	[0.02,0.03]	[4757,19116]	[9,10]
1.125	[5,23]	[0.087,0.116]	[0.02,0.04]	[4882,13666]	[10,10]
1.25	[5,23]	[0.085,0.111]	[0.01,0.03]	[4034,8950]	[10,10]
1.375	[5,23]	[0.09,0.092]	[0.02,0.02]	[3738,7749]	[10,10]
1.5	[5,23]	[0.089,0.095]	[0.01,0.03]	[3677,6096]	[10,10]
1.625	[5,23]	[0.09,0.095]	[0.01,0.02]	[3311,5619]	[10,10]
1.75	[5,23]	[0.089,0.097]	[0.01,0.02]	[2926,5605]	[10,10]
1.875	[5,23]	[0.091,0.112]	[0.01,0.02]	[2637,8068]	[10,10]
2	[5,23]	[0.087,0.111]	[0.01,0.02]	[2671,5708]	[10,10]
2.125	[5,23]	[0.092,0.123]	[0.01,0.02]	[2305,4280]	[10,10]
2.25	[5,23]	[0.091,0.098]	[0.01,0.02]	[2226,5855]	[10,10]
2.375	[5,23]	[0.088,0.105]	[0.01,0.02]	[2308,4661]	[10,10]
2.5	[5,23]	[0.089,0.099]	[0.01,0.02]	[2202,3479]	[10,10]

Continued on next page

Threshold	Hid. Neuron	ARD	Approx. dur.	Mean Steps	# of ANNs
2.625	[5,23]	[0.089,0.113]	[0.01,0.02]	[1877,3046]	[10,10]
2.75	[5,23]	[0.092,0.104]	[0.01,0.01]	[1570,3090]	[10,10]
2.875	[5,23]	[0.094,0.102]	[0.01,0.02]	[1751,4888]	[10,10]
3	[5,23]	[0.093,0.107]	[0.01,0.02]	[1718,2692]	[10,10]
3.125	[5,23]	[0.093,0.11]	[0.01,0.01]	[1769,2785]	[10,10]
3.25	[5,23]	[0.093,0.104]	[0.01,0.01]	[1496,3816]	[10,10]
3.375	[5,23]	[0.092,0.103]	[0.01,0.01]	[1527,2986]	[10,10]
3.5	[5,23]	[0.095,0.124]	[0.01,0.01]	[1168,2611]	[10,10]
3.625	[5,23]	[0.091,0.105]	[0.01,0.02]	[1255,2364]	[10,10]
3.75	[5,23]	[0.094,0.103]	[0.01,0.01]	[1464,2265]	[10,10]
3.875	[5,23]	[0.095,0.124]	[0.01,0.02]	[1212,2931]	[10,10]
4	[5,23]	[0.094,0.129]	[0.01,0.01]	[1272,2168]	[10,10]
4.125	[5,23]	[0.094,0.113]	[0.01,0.01]	[1212,3028]	[10,10]
4.25	[5,23]	[0.094,0.109]	[0.01,0.01]	[1021,4433]	[10,10]
4.375	[5,23]	[0.094,0.113]	[0.01,0.01]	[1185,2336]	[10,10]
4.5	[5,23]	[0.093,0.111]	[0.01,0.01]	[1235,1977]	[10,10]
4.625	[5,23]	[0.096,0.111]	[0.01,0.01]	[985,2047]	[10,10]
4.75	[5,23]	[0.093,0.114]	[0.01,0.01]	[876,2158]	[10,10]
4.875	[5,23]	[0.096,0.119]	[0.01,0.01]	[919,2270]	[10,10]
5	[5,23]	[0.093,0.103]	[0.01,0.01]	[930,2023]	[10,10]

Table 7 presents extreme values of evaluation results for two hidden layer ANN's. Values in square brackets represent the minimum and maximum values for all of the following hidden neuron configurations [(3,2),(5,3),(7,4),(9,5),(11,6),(13,7),(15,8)]. The column "approximate duration" reflects computation time for attempting to train 10 repetitions of the given architecture in days.

Table 7: Evaluation results of ANNs with two hidden layers

Threshold	Hid. Neuron	ARD	Approx. dur.	Mean Steps	# of ANNs
0.05	[(3,2),(15,8)]	[NA,NA]	[0.09,0.37]	[66377,100000]	[0,7]
0.075	[(3,2),(15,8)]	[NA,NA]	[0.07,0.26]	[52418,100000]	[0,9]
0.1	[(3,2),(15,8)]	[NA,NA]	[0.07,0.22]	[35515,100000]	[0,10]
0.125	[(3,2),(15,8)]	[0.085,0.09]	[0.07,0.14]	[20088,96771]	[3,10]
0.15	[(3,2),(15,8)]	[0.084,0.087]	[0.07,0.13]	[12528,98945]	[1,10]
0.175	[(3,2),(15,8)]	[0.084,0.089]	[0.07,0.09]	[12935,89341]	[5,10]
0.2	[(3,2),(15,8)]	[0.085,0.087]	[0.05,0.09]	[16048,82901]	[5,10]
0.225	[(3,2),(15,8)]	[0.086,0.09]	[0.04,0.08]	[8580,61265]	[9,10]
0.25	[(3,2),(15,8)]	[0.085,0.09]	[0.04,0.06]	[9686,71407]	[6,10]
0.275	[(3,2),(15,8)]	[0.083,0.089]	[0.04,0.05]	[7072,64904]	[8,10]
0.3	[(3,2),(15,8)]	[0.085,0.09]	[0.04,0.06]	[7346,60011]	[7,10]
0.325	[(3,2),(15,8)]	[0.084,0.087]	[0.03,0.05]	[6467,59098]	[8,10]
0.35	[(3,2),(15,8)]	[0.085,0.089]	[0.03,0.05]	[7174,55130]	[8,10]
0.375	[(3,2),(15,8)]	[0.086,0.09]	[0.03,0.04]	[5106,37384]	[9,10]

Continued on next page

Threshold	Hid. Neuron	ARD	Approx. dur.	Mean Steps	# of ANNs
0.4	[(3,2),(15,8)]	[0.087,0.093]	[0.02,0.05]	[4831,63226]	[7,10]
0.425	[(3,2),(15,8)]	[0.086,0.103]	[0.02,0.04]	[3425,33966]	[10,10]
0.45	[(3,2),(15,8)]	[0.086,0.089]	[0.02,0.04]	[4194,54708]	[7,10]
0.475	[(3,2),(15,8)]	[0.085,0.089]	[0.02,0.03]	[3539,44414]	[7,10]
0.5	[(3,2),(15,8)]	[0.086,0.093]	[0.02,0.03]	[3697,34804]	[10,10]
0.625	[(3,2),(15,8)]	[0.088,0.092]	[0.01,0.03]	[2170,34858]	[8,10]
0.75	[(3,2),(15,8)]	[0.088,0.095]	[0.01,0.02]	[1898,17255]	[10,10]
0.875	[(3,2),(15,8)]	[0.087,0.092]	[0.01,0.02]	[1876,18302]	[10,10]
1	[(3,2),(15,8)]	[0.087,0.097]	[0.01,0.02]	[2154,9560]	[10,10]
1.125	[(3,2),(15,8)]	[0.086,0.099]	[0.01,0.02]	[1275,18193]	[10,10]
1.25	[(3,2),(15,8)]	[0.088,0.094]	[0.01,0.02]	[1116,16083]	[10,10]
1.375	[(3,2),(15,8)]	[0.089,0.094]	[0.01,0.02]	[1177,6237]	[10,10]
1.5	[(3,2),(15,8)]	[0.086,0.093]	[0.01,0.01]	[1246,9776]	[10,10]
1.625	[(3,2),(15,8)]	[0.088,0.098]	[0.01,0.01]	[1022,4636]	[10,10]
1.75	[(3,2),(15,8)]	[0.089,0.1]	[0.01,0.02]	[970,16712]	[9,10]
1.875	[(3,2),(15,8)]	[0.087,0.095]	[0.01,0.01]	[1227,7278]	[10,10]
2	[(3,2),(15,8)]	[0.088,0.101]	[0.01,0.01]	[866,11826]	[10,10]
2.125	[(3,2),(15,8)]	[0.09,0.097]	[0.01,0.02]	[808,8356]	[10,10]
2.25	[(3,2),(15,8)]	[0.087,0.098]	[0.01,0.01]	[900,3919]	[10,10]
2.375	[(3,2),(15,8)]	[0.091,0.101]	[0.01,0.01]	[876,6191]	[10,10]
2.5	[(3,2),(15,8)]	[0.091,0.101]	[0.01,0.02]	[523,22745]	[8,10]
2.625	[(3,2),(15,8)]	[0.088,0.099]	[0.01,0.01]	[618,3579]	[10,10]
2.75	[(3,2),(15,8)]	[0.089,0.105]	[0.01,0.01]	[545,3923]	[10,10]
2.875	[(3,2),(15,8)]	[0.09,0.102]	[0.01,0.01]	[800,3345]	[10,10]
3	[(3,2),(15,8)]	[0.09,0.102]	[0.01,0.01]	[675,3744]	[10,10]
3.125	[(3,2),(15,8)]	[0.092,0.108]	[0.01,0.01]	[504,2917]	[10,10]
3.25	[(3,2),(15,8)]	[0.09,0.101]	[0.01,0.01]	[599,4630]	[10,10]
3.375	[(3,2),(15,8)]	[0.089,0.102]	[0.01,0.01]	[655,2114]	[10,10]
3.5	[(3,2),(15,8)]	[0.089,0.096]	[0.01,0.01]	[523,12764]	[9,10]
3.625	[(3,2),(15,8)]	[0.096,0.104]	[0.01,0.01]	[456,4377]	[10,10]
3.75	[(3,2),(15,8)]	[0.093,0.104]	[0.01,0.01]	[454,4625]	[10,10]
3.875	[(3,2),(15,8)]	[0.093,0.105]	[0.01,0.01]	[442,4190]	[10,10]
4	[(3,2),(15,8)]	[0.09,0.106]	[0.01,0.01]	[523,2525]	[10,10]
4.125	[(3,2),(15,8)]	[0.094,0.099]	[0.01,0.01]	[440,1699]	[10,10]
4.25	[(3,2),(15,8)]	[0.096,0.125]	[0.01,0.01]	[398,2158]	[10,10]
4.375	[(3,2),(15,8)]	[0.093,0.107]	[0.01,0.01]	[504,1493]	[10,10]
4.5	[(3,2),(15,8)]	[0.095,0.145]	[0.01,0.01]	[461,11878]	[9,10]
4.625	[(3,2),(15,8)]	[0.093,0.116]	[0.01,0.01]	[397,2367]	[10,10]
4.75	[(3,2),(15,8)]	[0.088,0.101]	[0.01,0.01]	[520,2214]	[10,10]
4.875	[(3,2),(15,8)]	[0.094,0.1]	[0.01,0.01]	[483,1714]	[10,10]
5	[(3,2),(15,8)]	[0.094,0.101]	[0.01,0.01]	[373,1651]	[10,10]