Neural Networks for Predicting Algorithm Runtime Distributions

Katharina Eggensperger, Marius Lindauer, Frank Hutter
University of Freiburg
{eggenspk, lindauer, fh}@cs.uni-freiburg.de

Abstract
Many state-of-the-art algorithms for solving hard combinatorial problems in artificial intelligence (AI) include elements of stochasticity that lead to high variations in runtime, even for a fixed problem instance. Knowledge about the resulting runtime distributions (RTDs) of algorithms on given problem instances can be exploited in various meta-algorithmic procedures, such as algorithm selection, portfolios, and randomized restarts. Previous work has shown that machine learning can be used to individually predict mean, median and variance of RTDs. To establish a new state-of-the-art in predicting RTDs, we demonstrate that the parameters of an RTD should be learned jointly and that neural networks can do this well by directly optimizing the likelihood of an RTD given runtime observations. In an empirical study involving five algorithms for SAT solving and AI planning, we show that neural networks predict the true RTDs of unseen instances better than previous methods, and can even do so when only few runtime observations are available per training instance.

1 Introduction
Algorithms for solving hard combinatorial problems often rely on random choices and decisions to improve their performance. For example, randomization helps to escape local optima, enforces stronger exploration and diversifies the search strategy by not only relying on heuristic information. In particular, most local search algorithms are randomized [Hoos and Stützle, 2004] and structured tree-based search algorithms can also substantially benefit from randomization [Gomes et al., 2000].

The runtimes of randomized algorithms for hard combinatorial problems are well-known to vary substantially, often by orders of magnitude, even when running the same algorithm multiple times on the same instance [Gomes et al., 2000; Hoos and Stützle, 2004; Hurley and O’Sullivan, 2015]. Hence, the central object of interest in the analysis of a randomized algorithm on an instance is its runtime distribution (RTD), in contrast to a single scalar for deterministic algorithms. Knowing these RTDs is important in many practical applications, such as computing optimal restart strategies [Luby et al., 1993], optimal algorithm portfolios [Gomes and Selman, 2001] and the speedups obtained by executing multiple independent runs of randomized algorithms [Hoos and Stützle, 2004].

It is trivial to measure an algorithm’s empirical RTD on an instance by running it many times to completion, but for new instances this is of course not practical. Instead, one would like to estimate the RTD for a new instance without running the algorithm on it.

There is a rich history in AI that shows that the runtime of algorithms for solving hard combinatorial problems can indeed be predicted to a certain degree [Brewer, 1995; Roberts and Howe, 2007; Fink, 1998; Leyton-Brown et al., 2009; Hutter et al., 2014]. These runtime predictions have enabled a wide range of meta-algorithmic procedures, such as algorithm selection [Xu et al., 2008], model-based algorithm configuration [Hutter et al., 2011], generating hard benchmarks [Leyton-Brown et al., 2009], gaining insights into instance hardness [Smith-Miles and Lopes, 2012] and algorithm performance [Hutter et al., 2013], and creating cheap-to-evaluate surrogate benchmarks [Eggensperger et al., 2018].

Given a method for predicting RTDs of randomized algorithms, all of these applications could be extended by an additional dimension. Indeed, predictions of RTDs have already enabled applications such as dynamic algorithm portfolios [Gagliolo and Schmidhuber, 2006b], adaptive restart strategies [Gagliolo and Schmidhuber, 2006a; Haim and Walsh, 2009], and predictions of the runtime of parallelized algorithms [Arbelaez et al., 2016]. To advance the underlying foundation of these applications, in this paper we focus on better methods for predicting RTDs. Specifically, our contributions are as follows:

1. We compare different ways of predicting RTDs and demonstrate that neural networks (NNs) can jointly predict all parameters of various parametric RTDs, yielding RTD predictions that are superior to those of previous approaches (which predict the RTD’s parameters independently).
2. We propose DistNet, a practical NN for predicting RTDs, and discuss the bells and whistles that make it work.
3. We show that DistNet achieves substantially better performance than previous methods when trained only on a few observations per training instance.
2 Related Work

The rich history in predicting algorithm runtimes focuses on predicting mean runtimes, with only a few exceptions. Hutter et al. (2006) predicted the single distribution parameter of an exponential RTD and Arbelaez et al. (2016) predicted the two parameters of log-normal and shifted exponential RTDs with independent models. In contrast, we jointly predict multiple RTD parameters (and also show that the resulting predictions are better than those by independent models).

The work most closely related to ours is by Gagliolo and Schmidhuber (2005), who proposed to use NNs to learn a distribution of the time left until an algorithm solves a problem based on features describing the algorithm’s current state and the problem to be solved; they used these predictions to dynamically assign time slots to algorithms. In contrast, we use NNs to predict RTDs for unseen problem instances.

All existing methods for predicting runtime on unseen instances base their predictions on instance features that numerically characterize problem instances. In particular in the context of algorithm selection, these instance features have been proposed for many domains of hard combinatorial problems, such as propositional satisfiability [Nudelman et al., 2004] and AI planning [Fawcett et al., 2014]. To avoid this manual step of feature construction, Loreggia et al. (2016) proposed to directly use the text format of an instance as the input to a NN to obtain a numerical representation of the instance. Since this approach performs a bit worse than manually constructed features, in this work we use traditional features, but in principle our framework works with any type of features.

3 ProblemSetup

The problem we address in this work can be formally described as follows:

Problem Statement (Predicting RTDs). Given

- a randomized algorithm $A$
- a set of instances $\Pi_{train} = \{\pi_1, \ldots, \pi_n\}$
- for each instance $\pi \in \Pi_{train}$:
  - $m$ instance features $f(\pi) = [f(\pi)_1, \ldots, f(\pi)_m]$
  - $k$ runtime observations $t(\pi) = [t(\pi)_1, \ldots, t(\pi)_k]$ obtained by executing $A$ on $\pi$ with $k$ different seeds,

the goal is to learn a model that can predict $A$’s RTD well for unseen instances $\pi_{n+1}$ with given features $f(\pi_{n+1})$.

Following the typical approach in the literature [Hutter et al., 2006; Arbelaez et al., 2016], we address this problem in two steps:

1. Determine a parametric family $D$ of RTDs with parameters $\theta$ that fits well across training instances;
2. Fit a machine learning model that, given a new instance and its features, predicts $D$’s parameters $\theta$ on that instance.

Figure 1 illustrates the pipeline we use for training these RTD predictors and using them on new instances.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Param.</th>
<th>PDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal (N)</td>
<td>$\mu, \sigma$</td>
<td>$\frac{1}{\sqrt{2\pi}\sigma^2} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$</td>
</tr>
<tr>
<td>Lognormal (LOG)</td>
<td>$s, \sigma$</td>
<td>$\frac{1}{\sqrt{2\pi}s^2} e^{-\frac{(\log(x)-\log(s))^2}{2s^2}}$</td>
</tr>
<tr>
<td>Exponential (EXP)</td>
<td>$\beta$</td>
<td>$\frac{1}{\beta} e^{-\frac{x}{\beta}}$</td>
</tr>
<tr>
<td>Inverse Normal (INV)</td>
<td>$\mu, \lambda$</td>
<td>$\left(\frac{\lambda}{2\pi}\right)^\frac{1}{2} e^{-\frac{(x-\mu)^2}{2\lambda^2}}$</td>
</tr>
</tbody>
</table>

Table 1: Considered RTD families

3.1 Parametric Families of RTDs

We considered a set of 4 parametric probability distributions (shown in Table 1 with exemplary instantiations shown in Figure 2), most of which have been widely studied to describe the RTDs of combinatorial problem solvers [Frost et al., 1997; Gagliolo and Schmidhuber, 2006a; Hutter et al., 2006].

First, we considered the Normal distribution (N) as a baseline, due to its widespread use throughout the sciences.

Since the runtimes of hard combinatorial solvers often vary on an exponential scale (likely due to the $\mathcal{NP}$-hardness of the problems studied), a much better fit of empirical RTDs is typically achieved by a lognormal distribution (LOG); this distribution is attained if the logarithm of the runtimes is normally-distributed and has been shown to fit empirical RTDs well in previous work [Frost et al., 1997].

Another popular parametric family from the literature on RTDs is the exponential distribution (EXP), which tends to describe the RTDs of many well-behaved stochastic local search algorithms well [Hoos and Stützle, 2004]. It is the unique family with the property that the probability of finding a solution in the next time interval (conditional on not having found one yet) remains constant over time.

By empirically studying a variety of alternative parametric families, we also found that an inverse Normal distribution (INV) tends to fit RTDs very well. By setting its $\lambda$ parameter close to infinity, it can also be made to resemble a normal distribution. Like LOG and EXP, this flexible distribution can model the relatively long tails of typical RTDs of randomized combinatorial problem solvers quite well.

3.2 Quantifying the Quality of RTDs

To measure how well a parametric distribution $D$ with parameters $\theta$ fits our empirical runtime observations $t(\pi) = (t(\pi)_1, \ldots, t(\pi)_k)$ (the empirical RTD), we use the likelihood $\mathcal{L}_D$ of parameters $\theta$ given all observations $t(\pi)$, which is equal to the probability of the observations under distribution $D$ with parameters $\theta$:

$$\mathcal{L}_D(\theta \mid t(\pi)_1, \ldots, t(\pi)_k) = \prod_{i=1}^{k} p_D(t(\pi)_i \mid \theta).$$  \hspace{1cm} (1)

Consequently, when estimating the parameters of a given empirical RTD, we use a maximum-likelihood fit. For numerical reasons, as is common in machine learning, we use the negative log-likelihood (NLLH) as a loss function to be minimized:

$$-\log\mathcal{L}_D(t(\pi)_1, \ldots, t(\pi)_k) = -\sum_{i=1}^{k} \log p_D(t(\pi)_i \mid \theta).$$    \hspace{1cm} (2)
Algorithm A

\[ J(W) = \frac{1}{|\Pi|} \sum_{\pi \in \Pi} - \log \mathcal{D}(\theta \mid t(\pi)_1, \ldots, t(\pi)_k) \cdot \max_{i \in \{1, \ldots, k\}} t(\pi)_i \] \hspace{1cm} (3)

\[ = -\frac{1}{|\Pi|} \sum_{\pi \in \Pi} \left( \sum_{i=1}^{k} \log p_D(t(\pi)_i | \theta) \right) + \log \max_{i \in \{1, \ldots, k\}} t(\pi)_i . \] \hspace{1cm} (4)

4 Joint Prediction of multiple RTD Parameters

Having selected a parametric family of distributions, the last part of our pipeline is to fit an RTD predictor for new instances as formally defined in Section 3. In the following, we briefly discuss how traditional regression models have been used for this problem, and why this optimizes the wrong loss function. We then show how to obtain better predictions with NNs and introduce DistNet for this task.

4.1 Generalizing from Training RTDs

A straightforward approach for predicting parametric RTDs based on standard regression models is to fit the RTD’s parameters \( \theta(\pi) \) for each training instance \( \pi \), and to then train a regression model on data points \((\hat{f}(\pi), \hat{\theta}(\pi))\) that directly maps from instance features to RTD parameters. There are two variants to extend these approaches to the problem of predicting multiple parameters of RTDs governed by \( p > 1 \) parameters (e.g. \( s \) and \( \sigma \) for LOG): (1) fitting \( p \) independent regression models, or (2) fitting a multi-output model with \( p \) outputs. These approaches have been used before based on Gaussian processes [Hutter et al., 2006], linear regression [Arbelaez et al., 2016] and random forests [Hutter et al., 2014; Hurley and O’Sullivan, 2015]

However, we note that these variants measure loss in the space of the distribution parameters \( \theta \) as opposed to the true loss in Equation (2) and that both variants require fitting RTDs on each training instance, making the approach inapplicable if we, e.g., only have access to a few runs for each of a thousands of instances. Now, we show how NNs can be used to solve both of these problems.

4.2 Predictions with Neural Networks

NNs have recently been shown to achieve state-of-the-art performance for many supervised machine learning problems as large data sets became available, e.g., in image classification and segmentation, speech processing and natural language processing. For a thorough introduction, we refer the interested reader to Goodfellow et al. (2016). Here, we apply NNs to RTD prediction.

Background on Neural Networks. NNs can approximate arbitrary functions by defining a mapping \( y = f(x; W) \) where \( W \) are the weights to be learnt during training to approximate the function. In this work we use a fully-connected feedforward network, which can be described as an acyclic graph that connects nonlinear transformations \( g \) in a chain, from layer to layer. For example, a NN with two hidden layers that predicts \( y \) for some input \( x \) can be written as:

\[ y = g^{\text{out}}(g^{(2)}(g^{(1)}(x W^{(1)}) W^{(2)}) W^{(3)}) , \] \hspace{1cm} (5)

with \( W^{(j)} \) denoting trainable network weights and \( g^{(j)} \) (the so-called activation function) being a nonlinear transformation applied to the weighted outputs of the \( j \)-th layer. We use the exp(\( \cdot \)) activation function for \( g^{\text{out}} \) to constrain all outputs to be positive.

NNs are usually trained with stochastic gradient descent (SGD) methods using backpropagation to effectively obtain gradients of a task-specific loss function for each weight.

Neural Networks for predicting RTDs. We have one input neuron for each instance feature, and we have one output neuron for each distribution parameter. To this end, we assume that we know the best-fitting distribution family from the previous step of our pipeline.

We train our networks to directly minimize the NLLH of the predicted distribution parameters given our observed runtimes. Formally, for a given set of observed runtimes and instance features, we minimize the following loss function in an end-to-end fashion:

\[ J(W) \propto - \sum_{\pi \in \Pi_{\text{train}}} \sum_{i=1}^{k} \log \mathcal{D}(\theta_W \mid f(\pi), t(\pi)_i). \] \hspace{1cm} (6)

\[ ^{1}\text{We ignore bias terms for simplicity of exposition.} \]
Here, $\theta_W$ denotes the values of the distribution parameters obtained in the output layer given an instantiation $W$ of the NN’s weights. This optimization process, which targets exactly our loss function of interest (Eq. (2)), allows to effectively predict all $p$ distribution parameters jointly. Since predicted combinations are judged directly by their resulting NLLH, the optimization process is driven to find combinations that work well together. This end-to-end optimization process is also more general as it removes the need of fitting an RTD on each training instance and thereby enables using an arbitrary set of algorithm performance data for fitting the model.

**DistNet: RTD predictions with NNs in practice.** Unfortunately, training an accurate NN in practice can be tricky and requires manual attention to many details, including the network architecture, training procedure, and other hyperparameter settings.

Specifically, to preprocess our runtime data $\langle f(\pi_i), t(\pi_i) \rangle_{i \in \mathbb{N}}$, we performed the following steps:

1. We removed all (close to) constant features.
2. For each instance feature type, we imputed missing values (caused by limitations during feature computation) by the median of the known instance features.
3. We normalized each instance feature to mean 0 and standard deviation 1.
4. We scaled the observed runtimes in a range of $[0, 1]$ by dividing it by the maximal observed runtime across all instances. This also helps the NN training to converge faster.

For training DistNet, we considered the following aspects:

1. Our first networks tended to overfit the training data if the training data set was too small and the network too large. Therefore we chose a fairly small NN with 2 hidden layers each with 16 neurons. In preliminary experiments we found that larger networks tend to achieve slightly better performance on our largest datasets, but we decided to strive for simplicity.
2. We considered each runtime observation as an individual data sample.
3. We shuffled the runtime observations (as opposed to, e.g., using only data points from a single instance in each batch) and used a fairly small batch size of 16 to reduce the correlation of the training data points in each batch.
4. Our loss function can have very large gradients because slightly suboptimal RTD parameters can lead to likelihoods close to zero (or a very large NLLH). Therefore, we used a fairly small initial learning rate of $1 \times 10^{-3}$ exponentially decaying to $1 \times 10^{-5}$ and used gradient clipping [Pascanu et al., 2014] on top of it.

Besides that, we used common architectural and parameterization choices: $\tanh$ as an activation function, SGD for training, batch normalization, and a $L2$-regularization of $1 \times 10^{-4}$. We call the resulting neural network DistNet.

### Table 2: Characteristics of the used data sets.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>#instances</th>
<th>#features</th>
<th>cutoff [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clasp-factorizing$^2$</td>
<td>2000</td>
<td>102</td>
<td>5000</td>
</tr>
<tr>
<td>Saps-CV-VAR$^2$</td>
<td>10011</td>
<td>46</td>
<td>60</td>
</tr>
<tr>
<td>Spear-QCP$^2$</td>
<td>8076</td>
<td>91</td>
<td>5000</td>
</tr>
<tr>
<td>YaleSAT-QCP$^2$</td>
<td>11747</td>
<td>91</td>
<td>5000</td>
</tr>
<tr>
<td>Spear-SWGCP$^2$</td>
<td>11182</td>
<td>76</td>
<td>5000</td>
</tr>
<tr>
<td>YaleSAT-SWGCP$^2$</td>
<td>11182</td>
<td>76</td>
<td>5000</td>
</tr>
<tr>
<td>LPG-Zenotravel$^3$</td>
<td>3999</td>
<td>165</td>
<td>300</td>
</tr>
</tbody>
</table>

### 5 Experiments

In our experiments, we study the following research questions:

**Q1** Which of the parametric RTD families we considered best describe the empirical RTDs of the SAT and AI planners we study?

**Q2** How do DistNet’s joint predictions of RTD parameters compare to those of popular random forest models?

**Q3** Can DistNet learn to predict entire RTDs based on training data that only contains a few observed runtimes for each training instance?

#### 5.1 Experimental Setup

We focus on predicting the RTDs of 5 well-studied algorithms, each evaluated on a different set of problem instances from two different domains:

- **Clasp-factorizing** is based on the tree-based CDCL solver Clasp [Gebser et al., 2012] which we ran on SAT-encoded factorization problems instances.
- **Saps-CV-VAR** is based on the local search SAT solver Saps [Hutter et al., 2002]. The SAT instances are randomly generated with a varying clause-variable ratio.
- **Spear-SWGCP/YalSAT-SWGCP** are based on the tree-search SAT solver Spear [Babić and Hutter, 2007] and on the local search SAT solver YaleSAT [Biere, 2014], a combination of different variants of ProbSAT [Balint and Schöning, 2012]. The instances are SAT-encoded small world graph coloring problems [Gent et al., 1999].
- **Spear-QCP/YalSAT-QCP** are based on the same solvers as Spear-SWGCP and YaleSAT-SWGCP. The SAT instances encode quasigroup completion instances [Gomes and Selman, 1997].
- **LPG-Zenotravel** is based on the local search AI-planning solver LPG [Gerevini and Serina, 2002]. The instances are from the zenotravel planning domain [Penberthy and Weld, 1994], which arise in a version of route planning.

To gather training data, we ran each algorithm (with default parameters) with 100 different seeds on each instance$^4$. This

$^2$Run on a compute cluster with nodes equipped with two Intel Xeon E5-2630v4 and 128 GB memory running CentOS 7.

$^3$Run on a compute cluster with nodes equipped with two Intel Xeon E5-2650v2 and 64 GB memory running Ubuntu 14.04.

$^4$We removed instances for which no instance features could be computed and only considered instances which could always be solved within a cutoff limit.
resulted in the 7 datasets shown in Table 2. We used the open-source neural network library keras [Chollet et al., 2015] for our NN, scikit-learn [Pedregosa et al., 2011] for the RF implementation, and scipy [Jones et al., 2001] for fitting the distributions. 5

5.2 Q1: Best RTD Families
Figure 3 shows some exemplary CDFs of our empirical RTDs; each line is the RTD on one of the instances. The different algorithms' RTDs show different characteristics with most instances having a long right tail and a short left tail. The RTDs of Clasp-factoring in contrast have a short left and right tail. Also, for some instances from Saps-CV-VAR and Spear-SWGCP the runtimes were very short and similar, causing almost vertical CDFs.

Table 3 shows a quantitative evaluation of the different RTD families we considered. Next to the normalized NLLH (see Equation (4)), we followed Arbelaez et al. (2016) and evaluated the Kolmogorov-Smirnov (KS) test as a goodness of fit statistical test. The KS-statistic is based on the maximal distance between an empirical distribution and the cumulative distribution function of a reference distribution. To aggregate the test results across instances, we count how often the KS-test rejected the null-hypothesis that our measured \( t(\pi_j) \) are drawn from a reference RTD.

Overall, the best fitted distributions closely resembled the true empirical RTDs, with a rejection rate of the KS-test of at most 15.2%. Hence, on most instances the best fitted distributions were not statistically significantly different from the empirical ones. For most scenarios the log-normal distribution (LOG) performed best, closely followed by the inverse Normal (INV). The Normal (N) and exponential (EXP) distributions performed worse for all scenarios. On Spear-SWGCP, the KS-test showed the most statistically significant differences for the best fitting distribution since the RTDs for some instances only have a small variance (see Figure 3) and cannot be perfectly approximated by the distributions we considered. Still, these distributions achieved good NLLH values.

5Code and data can be obtained from here: http://www.ml4aad.org/distnet

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>INV LOG N EXP</th>
<th>INV LOG N EXP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clasp-factoring</td>
<td>-0.35 -0.35 -0.29</td>
<td>12.0 10.2 15.0</td>
</tr>
<tr>
<td>Saps-CV-VAR</td>
<td>-0.88 -0.88 -0.75</td>
<td>0.1 4.0 20.1</td>
</tr>
<tr>
<td>Spear-QCP</td>
<td>-1.20 -1.14 -1.10</td>
<td>1.1 22.6 52.1</td>
</tr>
<tr>
<td>YalSAT-QCP</td>
<td>-0.78 -0.78 -0.66</td>
<td>6.8 46.5 80.1</td>
</tr>
<tr>
<td>Spear-SWGCP</td>
<td>-0.93 -0.90 -0.71</td>
<td>15.2 26.7 24.5</td>
</tr>
<tr>
<td>YalSAT-SWGCP</td>
<td>-0.94 -0.91 -0.89</td>
<td>25.3 14.0 98.0</td>
</tr>
<tr>
<td>LPG-Zenotravel</td>
<td>-0.90 -0.90 -0.62</td>
<td>12.7 20.2 79.2</td>
</tr>
</tbody>
</table>

Table 3: Results for fitted distributions: average NLLH across instances and percentage of rejected distributions according to a KS-test (\( \alpha = 0.01 \) without multiple testing correction). For each scenario, we report result for all distributions ranked by the NLLH. For both metrics, smaller numbers are better.

5.3 Q2: Predicting RTDs

Next, we turn to the empirical evaluation of our DistNet and compare it to previous approaches. Since random forests (RFs) have been shown to perform very well for standard runtime prediction tasks [Hutter et al., 2014; Hurley and O’Sullivan, 2015], we experimented with them in two variants: fitting a multi-output RF (mRF) and fitting multiple independent RFs, one for each distribution parameter (iRF). We trained DistNet as described in Section 4.2 and limit the training to at most 1h or 1000 epochs, whichever was less. As a gold standard, we report the NLLH obtained by a maximum likelihood fit to the empirical RTD ("fitted" in Table 3).

Table 4 shows the NLLH achieved using a 10-fold cross-validation, i.e., we split the instances into 10 disjoint sets, train our models on all but one subset and measure the test performance on the left out subset. We report the average performance (see Eq.(4)) on train and test data across all splits for the two best fitting distributions from Table 3.

Overall our results show that it is possible to predict RTD parameters for unseen instances, and that DistNet performed best. For 4 out of 7 datasets, our models achieved a NLLH close to the gold standard of fitting the RTDs to the observed data. Also, for 4 out of 7 datasets both distribution families were similarly easy to predict for all models.

For the RF-based models, we observed slight overfitting for most scenarios. For DistNet, we only observed this on the smallest data set: Clasp-factoring. On Spear-SWGCP, both the iRF and mRF yielded poor performance for both distributions as they failed to predict distribution parameters for instances with a very short runtime and thus receive a high NLLH on these instances. In general the iRF yielded worse performance than mRF demonstrating that distribution parameter should be learned jointly. Overall, DistNet yielded the most robust results. It achieved the best test set predictions for all cases, sometimes with substantial improvements over the RF baselines.


### 5.4 Q3: DistNet on a Low Number of Observations

Finally, we evaluated the performance of DistNet wrt. the number of observed runtimes per instance. Fewer observations per instance result in smaller training data sets for DistNet, whereas the data set size for the mRF stays the same with the distribution parameters being computed on fewer samples. We evaluated DistNet and mRF in the same setup as before, using a 10-fold cross-validation, but repeating each evaluation 10 times with a different set of sampled observations. Figure 4 reports the achieved NLLH for LOG as a function of the number of training samples for two representative scenarios. We observed similar results for all scenarios.

Overall, our results show that the predictive quality of mRF relies on the quality of the fitted distributions used as training data whereas DistNet can achieve better results as it directly learns from runtime observations. On LPG-Zenotravel, for which all models performed competitively when using 100 observations (see Table 4), DistNet achieved a better performance with fewer data converging to a similar NLLH value as the RF when using all available observations. On the larger dataset, *YalSAT-QCP*, DistNet converged with 16 samples per instance yielding a predictive performance better than that of mRFs with 100 samples. Collecting only 16 instead of 100 samples would speed up the computation by more than 6-fold.

### 6 Conclusion and Future Work

In this paper we showed that NNs can be used to jointly learn distribution parameters to predict RTDs. In contrast to previous RF models, we train our model on individual runtime observations, removing the need to first fit RTDs on all training instances. More importantly, our NN – which we dub *DistNet* – directly optimizes the loss function of interest in an end-to-end fashion, and by doing so obtains better predictive performance than previously-used RF models that do not directly optimize this loss function. Because of that our model can learn meaningful distribution parameters from only few observations per training instance and therefore requires only a fraction of training data compared to previous approaches.

Overall, our methodology allows for better RTD predictions and therefore may pave the way for improving many applications that currently rely mostly on mean predictions only, such as, e.g., algorithm selection and algorithm configuration.

Currently, our method assumes large homogeneous instance sets without censored observations. We consider further extensions as future work, such as handling censored observations (e.g., timeouts) in the loss function of our NN [Gagliolo and Schmidhuber, 2006a, using a mixture of models [Jacobs et al., 1991] to learn different distribution families, or studying non-parametric models (which can fit arbitrary distributions and thus do not require prior knowledge of the type of runtime distribution). Finally, in many applications the algorithm’s configuration is a further important dimension for predicting RTDs, and we therefore plan to handle this as an additional input in future versions of DistNet.

### Acknowledgements

The authors acknowledge funding by the German Research Foundation (DFG) under Emmy Noether grant HU 1900/2-1 and support by the state of Baden-Württemberg through bwHPC and through grant no INST 39/963-1 FUGG. K. Eggensperger additionally acknowledges funding by the State Graduate Funding Program of Baden-Württemberg.
References


